WEST

Generate Collection Print

L25: Entry 19 of 34 File: USPT Jan 15, 2002

DOCUMENT-IDENTIFIER: US 6339822 B1
TITLE: Using padded instructions in a block-oriented cache

Brief Summary Text (10):
Caches may be configured in a number of different ways. For example, many caches are set-associative, meaning that a particular line of instruction bytes may be stored in a number of different locations within the array. In a set-associative structure, the cache is configured into two parts, a data array and a tag array. Both arrays are two-dimensional and are organized into rows and columns. The column is typically referred to as the "way." Thus a four-way set-associative cache would be configured with four columns. A set-associative cache is accessed by specifying a row in the data array and then examining the tags in the corresponding row of the tag array. For example, when a prefetch unit searches its instruction cache for instructions residing at a particular address, a number of bits from the address are used as an "index" into the cache. The index selects a particular row within the data array and a corresponding row within the tag array. The number of address bits used for the index are thus determined by the number of rows configured into the cache. The tags addresses within the selected row are examined to determine if any match the requested address. If a match is found, the access is said to be a "hit" and the data cache provides the associated instruction bytes from the data array. If a match is not found, the access is said to be a "miss." When a miss is detected, the prefetch unit causes the requested instruction bytes to be transferred from the memory system into the data array. The address associated with the instruction bytes is then stored in the tag array.

Brief Summary Text (12):
After a requested instruction address is output to main memory, a predetermined number of sequential instruction bytes beginning at the requested address are read from main memory, predecoded, and then conveyed to the instruction cache for storage. The instruction bytes are stored into storage locations ("cache lines") according to their address, typically without regard to what types of instructions are contained within the sequence of instruction bytes.

Brief Summary Text (13):
One drawback, however, of traditional caches is that they suffer from inefficiencies because branch instructions and branch targets do not naturally occur at cache line boundaries. This may deleteriously affect performance because taken branch instructions residing in the middle of a cache line may cause the end portion of the cache line to be discarded when it is fetched. Furthermore, branch targets that are not located at the start of a cache line may similarly cause the beginning portion of the cache line to be discarded. For example, upon receiving a fetch address, the typical instruction cache reads the entire corresponding cache line, and then selection logic (either internal or external to the instruction cache) selects the desired instructions and discards instruction bytes before the target address and or after a branch instruction.

Brief Summary Text (19):
A microprocessor configured to cache basic blocks of instructions is also contemplated. In one embodiment, the microprocessor comprises a basic block cache and a basic block sequence buffer. The basic block cache is configured to store basic blocks, wherein each basic block may comprise a number of instructions and may end with a branch instruction. The basic block sequence buffer comprises a plurality of storage locations, each configured to store a block sequence entry. The block

sequence entry has an address tag and one or more basic block pointers. The address tag corresponds to the fetch address of a particular basic block, and the pointers point to basic blocks that follow that particular basic block in a predicted order. Each block sequence entry may contain multiple basic block pointers and branch prediction information to select the basic block that is predicted to follow the block corresponding to the address tag.

Drawing Description Text (13):
FIG. 10B is a diagram illustrating one possible method for storing information about basic blocks.

Detailed Description Text (14):
BBSB 42 may be configured to have the same structure (e.g., addressing scheme) as BBC 44 and to receive the same fetch address information that BBC 44 receives from decode unit 20. However, instead of storing basic blocks, BBSB 42 is configured to store information about the corresponding basic blocks in BBC 44. For example, BBSB 42 may store predicted sequences of basic blocks and the addresses of all possible following basic blocks. It may also contain prediction information indicative of whether the corresponding branch instructions (that define the end of each basic block) will be taken or not taken. This prediction information may be used to select which basic block will be executed next.

Detailed Description Text (27):
In one particular embodiment of microprocessor 10 employing the x86 microprocessor architecture, instruction cache 16 and data cache 28 are linearly addressed. The linear address is formed from the offset specified by the instruction and the base address specified by the segment portion of the x86 address translation mechanism. Linear addresses may optionally be translated to physical addresses for accessing a main memory. The linear to physical translation is specified by the paging portion of the x86 address translation mechanism. It is noted that a linear addressed cache stores linear address tags. A set of physical tags (not shown) may be employed for mapping the linear addresses to physical addresses and for detecting translation aliases. Additionally, the physical tag block may perform linear to physical address translation

Detailed Description Text (35):
The form of the pointers provided by BBSB 42 and the method of detecting hits in BBC 44 is subject to the same tradeoff as conventional branch prediction, where there are two basic approaches. The first approach is for the predictor to provide a full target address for the cache lookup. A normal cache tag comparison is then performed to determine if the block is in the cache. An alternate approach is to store only cache block addresses in the predictor array (i.e., a particular cache block is predicted to contain the proper target instructions), thereby greatly reducing the array size. The full address is formed from the block index and the cache tag, and then sent to functional units 24A-N for verification. Note that verification may be performed for either scheme. With this method a cache miss may not be detected until the branch in question is executed, hence there may be a tradeoff of less real estate versus greater delay in detecting cache misses.

Detailed Description Text (36):
Turning now to FIG. 4, another embodiment of BBC 44 is shown. In this embodiment, BBSB 42 and BBC 44 are accessed in parallel with the fetch address during the same clock cycle. Since BBC 44 may output the corresponding basic block as soon as it is available (versus waiting a clock cycle as in the previous embodiment), the first basic block (BBn) may be available one clock cycle sooner. This is reflected in the output as indicated in Table 2.

Detailed Description Text (47):
This section describes one possible method for addressing a line within BBSB 42. Depending upon the implementation, BBSB 42 may be addressed with either physical or linear addresses. In some embodiments, the addressing of BBSB 42 may differ from that of a regular cache. For example, a regular cache may have a fixed line size, e.g., 32 bytes. Thus, the index into the cache addresses 32 byte multiples. Accordingly, the index used to access the cache is the upper portion of the address less the offset. In this way, two sequential lines may each start with an offset of

zero, but with an index which differs by one.

Detailed Description Text (55):
Turning now to FIG. 10A, a table of sample addresses of a basic block sequence is shown. The sample addresses illustrate a number of short (i.e., two byte) sequential basic blocks. One possible method for storing information about these basic blocks is illustrated in FIG. 10B. The figure shows the basic block information stored in a four-way set associative embodiment of BBSB 42. As the figure illustrates, the first basic block in the sequence is stored in way 0, the second basic block in way 1, the third basic block in way 2, and the fourth basic block in way 3. The set is selected by the index portion of the address of each basic block, in this case 00.

Detailed Description Text (58):
Each line of BBSB 42 may contain information about a particular basic block sequence. As previously discussed, a sequence of two basic blocks may result in the storage of six basic block addresses and prediction information for three branches. In one embodiment, BBSB 42 stores full 32-bit addresses fore each of the basic blocks. Thus an exemplary storage line within BBSB 42 may contain the fields illustrated in FIG. 11.

Detailed Description Text (98):
BBC 44 accesses and attempts to output cache lines corresponding to the two addresses from both ports. Assuming both addresses hit in BBC 44, the following information is provided to reorder buffer 32: basic block 1 (instructions 1-4), basic block 2 (instructions 1-4), the valid bits for each instruction in each basic block, and possibly additional predecode and reorder buffer (i.e., dependency) information. At the end of the clock cycle, the replacement information within BBC 44 may also be updated.

Detailed Description Text (120):
When a miss occurs, instruction cache 16 may be looked up with the missed address from BBSB 42. The missed address may be either from the first basic block or the second basic block. If both basic blocks miss, then the first basic block may be fetched first. Assuming the missing address hits in instruction cache 16, the instructions may be directly fed to decode unit 20.

Detailed Description Text (130):
Next, a determination may be made as to whether the mispredicted basic block was the second basic block in a BBSB entry or the starting basic block of the next BBSB entry. If it was the second basic block, then the BBC is looked up with the new address of the basic block. Assuming a hit occurs, at the end of the clock cycle the basic block will be available to reorder buffer 32. From there, the instructions step through the final parts of the pipeline. Note that in this case only four instructions (I basic block) are dispatched. The other instructions are NULL instructions. Next, the predictor for the third basic block is used to access BBSB 44 again. Instead, if it was the starting basic block of the next BBSB entry, then BBSB 42 is looked up using the new address.

Detailed Description Text (133):
In one embodiment, BBSB 42 may be configured to send the following information about the basic block sequence to branch prediction unit 14: (1) the predicted address of next basic block, (2) the pointer to the BBSB entry, and (3) an identifier of the branch path with the basic block tree (e.g., BB1_0, BB2_0, BB2_1).

Detailed Description Text (141):
For example, in one embodiment, if a basic block is the second basic block in a sequence, then the basic block address may not be easily looked up in BBSB 42. Only the start address of the first basic block may be looked up. In addition, multiple BBSB entries might point to the same BBC entry for the second basic block. Therefore, it may be difficult to ensure that all BBSB entries are invalidated for a given basic block address.

Detailed Description Text (145):
Using the invalidation window, each line in BBC 44 is invalidated if it is selected with the index and the UP_TAG matches the INV_ADR. The LOW_TAG field is not

considered. Thus, for invalidation the basic blocks start at an address where the lowest three address bits (A2-A0) are zero. For example, for a particular INV_ADR all instructions in the range from INV_ADR up to INV_ADR +31 may be invalidated.

Detailed Description Text (153):
In the event that the basic block starts at an address below the INV_ADR, it may extend into the invalidation window. To address this possibility, these instructions may also be invalidated. In one embodiment this is accomplished by subtracting the maximum basic block length from the INV_ADR as indicated by the following equation: (INV_ADR_LOW) =(INV-ADR)--(Maximum Basic Block Length). The generation of INV_ADR_LOW is illustrated in FIG. 20. INV_ADR_LOW may be used instead of INV_ADR to invalidate the BBC entries. This process may be performed in a similar fashion to that of case one above, except for using INV_ADR_LOW instead of INV_ADR.

Detailed Description Text (154):
Assuming for explanatory purposes that the maximum basic block length 60 bytes long (i.e., 4 x maximum instruction length=4.times.15 byte=60 bytes), using an index resolution of eight bytes would result in eight invalidation cycles. This may, however, be reduced if the maximum basic block size is limited to a smaller size, e.g., 32 bytes. The process of limiting the maximum basic block length is described below. Assuming an embodiment of microprocessor 10 has limited the maximum basic block size length to 32 bytes, then the invalidation process may be reduced to only 4 invalidation cycles (index 0-3).

Detailed Description Text (159):
As the above embodiments indicate, invalidations may deleteriously affect performance if they occur too frequently. Some embodiments may, however, elect to invalidate more instructions than necessary in order to simply the implementation. This is true for instructions within a basic block that begins in the invalidation window but extends outside the invalidation window. In this case the end instruction may be unnecessarily invalidated. In some cases, entire basic blocks may be invalidated unnecessarily. This may be the case with a basic block that is smaller than the maximum basic block size. This is illustrated in FIG. 22. If the block starts after INV_ADR_LOW but ends before INV_ADR, then it may be unnecessarily invalidated.

Detailed Description Text (160):
However, even with unnecessary invalidations, cache coherency is still maintained. Thus, no false instructions are processed. The only potential negative effect may be on performance because additional unnecessary BBC misses may occur. In one embodiment, microprocessor 10 may be configured to detect and avoid these unnecessary invalidations. For example, microprocessor 10 may be configured with invalidation hardware that uses the basic block length information to generate the ending addresses of the basic blocks. Then bounds checking hardware may be used to determine whether the basic block needs to be invalidated or not. However, this more complicated hardware may not be necessary given the relatively low frequency in which such unnecessary invalidations may occur in many implementations.

Detailed Description Text (166):
After the above BBSB miss sequence has been executed, there may still be one additional case to consider. Assuming that all the BBC entries have been filled with valid basic block information, other BBSB entries may point to the changed BBC entries as their second or third basic block. If an access to the entry that points to a changed entry results in a BBC miss, it will trigger the BBSB miss process described above that will re-synchronize the changed entries.

CLAIMS:

11. The microprocessor as recited in claim 10, wherein said decoding logic is further configured to generate and store operand register dependency information with each basic block in said basic block cache.

17. A method of operating a cache within a microprocessor comprising:

receiving instruction bytes corresponding to a fetch address;

decoding said instruction bytes into instructions;

padding the instructions to one of a plurality of predetermined instruction lengths;

dispatching the instructions for execution;

grouping the padded instructions into basic blocks that end with branch instructions;

storing the basic blocks into a basic block cache;

storing link information for each cache line within the basic block cache, wherein the link information is indicative of whether each particular basic block fits within a single cache line; and

storing branch prediction information for the basic blocks into a branch prediction array.

WEST

Generate Collection Print

L25: Entry 13 of 34 File: USPT Jul 16, 2002

DOCUMENT-IDENTIFIER: US 6421279 B1
TITLE: Flash memory control method and apparatus processing system therewith

Brief Summary Text (28):
On the other hand, high-performance personal computers, etc., often use a DRAM-SRAM cache system as means for shortening the read or write time. Generally, the cache memory is located between the CPU and storage taking time to access for serving as a buffer memory. When the CPU reads the storage, the read address and data are stored in the cache memory. When the CPU then reads the same read address of the storage, the data corresponding to the address is obtained from the cache memory, thereby shortening the access time. The two systems of cache memory are known: Write through and copy back. The write through system is a system which rewrites the storage as well as the cache memory at the same time in response to a write request into a storage. On the other hand, the copy back system is a system which is responsive to a write request into a storage for rewriting only the cache memory without rewriting the storage which requires a lot of processing time and is intended to shorten the access time.

Brief Summary Text (29):
The cache memory system generally used with information processing systems such as personal computers at present includes the main memory of DRAM (dynamic random access memory) and a cache memory of SRAM (static random access memory) to cover the weak point that the DRAM access operation cannot keep up with the CPU operation speed. Accessed addresses are allocated to the SRAM and the DRAM accessed at slow speed is used to back up data as if the SRAM accessed at fast speed were the main memory when viewed from the CPU. In this technique, the SRAM access speed is several times as fast as the DRAM access speed, and although it is less than ten times as fast. Thus, when a write access is made to an address not allocated to the cache memory, namely, when write miss occurs, the recovery time is not so great. If the flash memory is adopted as the main memory, the flash memory has the rewrite time 1000 to 100000 times longer than the DRAM, and the recovery time at write miss becomes very great, lowering system performance. Therefore, this point must be considered when implementing a system.

Brief Summary Text (71):
To write data, the host system sends a write instruction together with address information (block identification information if data is written into each block) to the semiconductor memory section. When receiving the write instruction via the interface circuit, the control circuit of the semiconductor memory section reads the write address of the semiconductor memory section corresponding to the given address information from the memory block management table, and writes the given data into the target area (block) of the data memory. When the control circuit detects that an error occurs in the block at time of writing, the memory management means reads address information of an unused block of the alternate memory section, allocates it as an alternate block, and sets information indicating that the alternate block is used. The control circuit writes the data into the alternate block.

Brief Summary Text (74):
To read data in the target block from the semiconductor disk, the semiconductor memory area corresponding to the address information for the data to be read is read from the memory management means and the data is read from the target block of the storage section. If an error occurs in the block, address information of the alternate area to the block is read from the memory management means and the data is

read from the alternate area.

Brief Summary Text (80):
An address array for recording addresses of data stored in the cache memory and storage means for recording an access history to the cache memory are provided.

Brief Summary Text (87):
As described above, the address array for recording logical addresses of data stored in the cache memory and the storage area for recording an access history indicating oldness of data stored in the cache memory are provided whereby a determination can be made as to whether or not one address existing in the cache memory is accessed. If an address not existing in the cache memory is accessed, the access history is searched for the data least accessed since the last access occurs and the data is written back into the flash memory, the main memory to create an empty area in the cache memory in which new data is stored. This is known as a cache memory replacement algorithm.

Brief Summary Text (88):
When a data write request is received from the CPU, if the address corresponding to the data is not stored in the cache memory, large performance degradation occurs in the slow write operation flash memory if an empty area for storing the write data is created after the request is received. Then, an empty area is always reserved in t he cache memory and the data is temporarily stored in the reserved empty area. After the write data from the CPU has been transferred, a step of creating an empty area in the flash memory may be started.

Drawing Description Text (84):
FIG. 82 is a schematic block diagram of a fourth embodiment of an information processing system using a flash memory as a main memory;

Drawing Description Text (85):
FIG. 83 is a block diagram of a controller in the information processing system using the flash memory as the main memory in FIG. 82;

Drawing Description Text (105):
FIG. 103 is a flowchart showing an operation flow of restoring address array data and cache memory data by the controller in the example system in FIG. 101.

Detailed Description Text (38):
Next, the operation if the microcomputer writes or reads data into or from PSRAM when it is being refreshed is described. PSRAM refresh and PSRAM write or read from the microcomputer are executed in the same bus cycle by extending the micromputer bus cycle. By the way, at power on reset, 10.phi. clock output of the microcomputer and 10.phi. clock provided by dividing the system clock for generating the refresh control signal 1179 may be out of phase. Thus, the timing relationship between the write or read control signal and the refresh control signal 1179 and out-of-phase of the clocks must be considered to determine how many clocks are to be extended. Therefore, when the microcomputer accesses PSRAM, the block 1175 determines out-of-phase state of the clocks and sends information to the block 1181, then the block 1181 outputs PSRAM write or read and refresh control signals and a wait signal (WAITN) 1180 for extending the microcomputer bus cycle.

Detailed Description Text (70):
The controller 2004 uses the degradation degree when the physical address is determined to store data. That is, data write into blocks whose degradation degree is judged to be great is avoided as much as possible, thereby preventing performance from lowering due to degradation. A flowchart showing the control sequence is discussed with reference to FIG. 58. FIG. 58 is a flowchart of measuring the write time to diagnose the degradation degree in a flash memory write operation. The diagnosis is made based on the fact that flash memory has a feature that the time required for writing is prolonged as degradation is advanced. The flash memory write routine in FIG. 58 shows writing into one erased block. When the host makes a write access and write data is stored in the write buffer, the routine is started. First, a block having the lowest degradation degree is looked up in the degradation degree information table at step 9a. If all blocks are at the same degradation level, any

desired block is selected. As a result, all blocks will degrade evenly. When a write block is found, the write time measurement circuit is started for each write unit to start measuring the write time, and at the same time, actual flash memory write is started at step 9b. A wait is made until the memory write terminates at step 9c. Upon completion of the write, the time required for the write is read from the write time measurement circuit and the degradation degree is diagnosed at step 9d. If the degradation degree diagnosis result is the worst among the write units written so far in the single block, the result is stored in the degradation degree information table at step 9e. In the flash memory in which a plurality of writes form one erasure block, one piece of the degradation degree information is provided for one erasure block. The degradation degree is diagnosed in each write unit and the worst value in the single block is judged as the degradation degree of the block. The degradation degree is considered to differ for each bit, and even if one bit degrades, reliability of the entire area in the block lowers. However, for simplification of the control program, the time only at a specific point in a block may be measured to determine the entire degradation degree.

Detailed Description Text (83):
FIG. 62 shows an example of the correspondence between a memory address map of the semiconductor memory 3106 and the memory block management table 3112. The memory block management table 3112 is memory management means for retaining information as to whether or not each block of the semiconductor memory is used in a block use table, and when the control circuit detects an error, for assigning an unused block as an alternate block in place of the error incurring block of the semiconductor memory and retaining the correspondence between the assigned alternate block and the error incurring block in a block registration table. As shown in the memory address map 3201, the 30M-byte data memory section 3108 comprises a data memory area 3202 (30M-byte space from address 0000000H (H denotes hexadecimal notation) to address 1DFFFFFH) and the 2M-byte alternate memory section 3109 comprises an alternate memory area 3203 (2M-byte space from addresses 1E00000H to 1FFFFFFH). Since data is written in 512-byte units in the embodiment, one block contains 200H addresses. For example, block 0 ranges from address 0000000H to address 00001FFH. Likewise, block 1 ranges from address 0000200H to address 00003FFH, block 2 ranges from address 0000400H to address 00005FFH, block 3 ranges from address 0000600H to address 00007FFH, . . . , as shown in the memory address map 3201. Address 1E00000H and later are assigned to alternate memory blocks in the same manner. As described above, assignment of the addresses is not unique and the addresses may be assigned in any desired manner.

Detailed Description Text (84):
The memory block management table comprises a block registration table 3205 for registering semiconductor memory addresses corresponding to blocks, a memory block use table 3206 for registering information as to whether or not each block of the data memory section 3108 is used, and an initialization information area 3207 for registering initialization information of the entire system.

Detailed Description Text (85):
The formats of the block registration table 3205 and the memory block use table 3206 are as shown in FIG. 62. The block registration table 3205, which lists addresses of the semiconductor memory 3106 corresponding to blocks, has a capacity of four bytes (32 bits) per entry of one block. The block registration table 3205 starts at address 2000000H and represents one block every 4H addresses. Block 0 is indicated by the address information stored in the 4-byte entry 3216 starting at address 2000000H. Likewise, block 1 is indicated by the address information stored in the 4-byte entry 3217 starting at address 2000004H and block 2 is indicated by the address information stored in the 4-byte entry 3218 starting at address 2000008H.

Detailed Description Text (86):
The block use table 3206 stores information as to whether or not each block of the data memory section 3108 and the alternate memory section 3109 is used. The use state of one block is represented by 1-bit information; in the embodiment, an unused block is represented as 0 and a used block as 1. An empty block in the alternate memory section 3109 can be found by searching the block use table for a "0" bit indicating an unused block. The block use table 3206 starts at address 2020000H and represents the use state of eight blocks per 1-byte use information entry. The least

significant bit of one byte represents the block having the smallest block number. That is, the 1-byte use information 3214 at address 2020000H represents the use state of eight blocks from blocks 0 to 7. For example, if the bit sequence of the one byte is 11011111b (b denotes binary notation), it indicates that only block 5 is unused. The region from addresses 2020000H to 2021DFFH represents the use state of the data memory section 3108 and the region from addresses 2021E00H to 2021FFFH represents the use state of the alternate memory section 3109.

Detailed Description Text (87):
Further, FIG. 77 shows a specific example of information stored in the initialization information area 3207. In FIG. 77, the initialization information area 3207 is an area which stores initialization information such as start address information 3231 of the data memory area 3202, end address information 3232 of the data memory area 3202, data memory area capacity 3233, storage capacity per block 3234, the number of available blocks 3235, start address information 3236 of the alternate memory area 3203, end address information 3237 of the alternate memory area 3203, and reserved information area 3238. Necessary information is written into the initialization information area 3207 when the entire disk system is initialized.

Detailed Description Text (89):
Next, the block registration table 3205 and the block use table 3206 are initialized at step 3703. First, address information corresponding to each disk block is written into the block registration table 3205. For example, to initialize block 0, address information 0000000H corresponding to block 0 of the data memory area 3202 is written into the 4-byte entry starting at address 2000000H of the block registration table 3205 corresponding to the block 0. Likewise, to initialize block 1, address information 0000200H corresponding to block 1 of the data memory area 3202 is written into the 4-byte entry starting at address 2000004H of the block registration table 3205. The operation is repeated for all blocks of the data memory area 3202 and the alternate memory area 3203 at step 3704. Further, if necessary, the error information register 3105 and the buffer memory 3115 are initialized. The initialization of the entire disk is now complete. The initialization operation only needs to be executed when the semiconductor disk unit 3102 is first used or when a disk format instruction is executed.

Detailed Description Text (94):
FIG. 63 shows a read process sequence of the semiconductor disk control circuit 3111. As shown in FIG. 63, the semiconductor disk control circuit 3111 reads address information of the semiconductor memory 3106 corresponding to the block number received from the I/O bus 3104 from the block registration table 3205 of the memory block management table 3112 at step. 3301. For example, to read block 0, 4-byte address information 3216 starting at address 2000000H of the block registration table 3205; to read block 1, 4-byte address information 3217 starting at address 2000004H of the block registration table 3205. Next, based on the address information read at step 3101, 512-byte information is read from the region corresponding to the block number of the data memory section 3108, for example, if block 0 is read, address 0000000H indicated by the address information 3216 at step 3302. The data is temporarily transferred to the buffer memory at step 3303. Then, the data is transferred via the interface circuit 3107 to the I/O bus 3104 at step 3304.

Detailed Description Text (101):
When a write error occurs, the semiconductor disk control circuit 3111 generates an interrupt signal by information means and stores information on the error in the error information register 3105 as shown in FIG. 79. As the stored error information, a bit indicating whether or not an error occurs and a bit indicating no empty alternate blocks are set when no unused blocks exist in the alternate memory area 3203 as described above. When no empty data blocks exist in the data memory area 3202, a bit indicating whether or not an error occurs and a bit indicating no empty data blocks are set. For the error incurring block, address information such as the block number of the block can be set in a block number registration field of the error information register 3105.

Detailed Description Text (102):

The error information register may be divided into an error information type field, reserved information field, and block number registration fields 1 and 2, as shown in FIG. 81. When an error occurs, the semiconductor disk control circuit 3111 sets error information as described above.

Detailed Description Text (109):
In FIG. 66, first the block use table 3206 of the memory block management table 3204 shown in FIG. 62 is searched for an unused block of the data memory section 3108 at step 3601. Specifically, the region at addresses 2020000H to 2021DFFFH of the block use table 3206 is searched for a bit set to 0. In the embodiment, the sixth least significant bit of the 1-byte information 3214 at address 2020000H is 0. As described above, it means that the sixth block of the data memory area 3202, namely, the 512-byte block 3210 starting at address 0000A00H is an unused block. Next, whether or not the data memory section 3108 contains an unused block is checked at step 3602. Since the block 3210 exists as an empty block in the embodiment, the bit 3220 of the block use table 3206 corresponding to the block 3210 is set to 1 (used) at step 3603. Next, at step 3604, the address information 3221 of the found empty block of the data memory area 3202 is written into the 4-byte entry starting at address 200000EH of the block registration table 3205 in FIG. 62 corresponding to the block 3210 into which data is to be written. In the embodiment, to write data into block 5, address information 0000A00H is written into address 200000EH of the block registration table 3205. Subsequently, block erasure is executed for the empty block of the data memory section 3108 at step 3605 and the contents of the buffer memory 3115 are written into the empty block 3210 at step 3606.

Detailed Description Text (110):
By the way, the block 3210 of the data memory section 3108 used as the alternate write area at the above-mentioned steps is a block originally used as a data area. Thus, there is a chance that the host system 3101 will issue another write instruction into the block 3210 of the data memory section 3108. Then, information such as the block number of the empty block of the data memory section 3108 used as the alternate area is written into the error information register 3105 at step 3607 and an interrupt signal 3103 is output to the host system 3101 at step 3608. When acknowledging the interrupt signal 3103, the host system 3101 may interrupt the current processing, read the block number contained in the error information register 3105, and perform proper processing such as inhibiting use of the block.

Detailed Description Text (112):
The reconfiguration operation of the data memory section and the alternate memory section as a processing method at the semiconductor disk control circuit in the embodiment is described with reference to a flowchart shown in FIG. 76. In the embodiment, when a write error occurs, an empty block is found in the alternate memory section 3109 and data is written into the found block, as described above. When there are no empty blocks of the alternate memory section 3109, empty blocks can be found in the data memory section 3108 to reconfigure the semiconductor disk unit 3102. To do this, in FIG. 76, first at step 3181, initialization information is read from the memory block management table 3112 shown in FIG. 62 and the block use table is searched for "0" bits for the data memory section 3108 to find unused blocks at step 3182 until the end of the table for the data memory section is reached at step 3183.

Detailed Description Text (113):
At step 3184, a check is made to see if empty blocks exist. If no empty blocks exist, the user is informed that no empty area exists at step 3188. If one or more empty blocks exist, the blocks are newly allocated to the alternate memory section 3109 and new initialization information is written into the initialization information area of the memory block management table 3112 at step 3185. Further, the blocks allocated to the alternate memory section are reported to the host system through the error information register 3105 for inhibiting use of the blocks at step 3186. Then, the user is informed that disk reconfiguration is complete at step 3187. The reconfiguration process is now complete.

Detailed Description Text (115):
Next, an example of the semiconductor disk system according to the third embodiment is discussed. In the example, the data memory section 3108 and the alternate memory

section 3109 of the semiconductor memory 3106 are mixed as a mixed data memory 3801, as shown in FIG. 68. FIG. 69 shows an example of the correspondence between a memory map of the mixed data memory 3801 shown in FIG. 68 and memory block management table 3112. In FIG. 69, as shown in the memory address map 3201, a mixed data area 3901 has data blocks and alternate blocks mixed and has a capacity of 32M bytes in total in the range of addresses 0000000H to 1FFFFFFH. In the example, the 30M-byte space is actually used as a data area and the remaining 2M-byte space is used as an alternate block area. The capacities of these two areas are not fixed and can be changed by the user who sets proper values in the initialization information area when the semiconductor disk unit is initialized. How to determine the capacities of the initialization information area 3207, block registration table 3205, and block use table 3206 is the same as in the example in FIG. 62.

Detailed Description Text (120):
Next, a fourth example of the semiconductor disk system according to the third embodiment is discussed, in which the alternate memory section 3109 in FIG. 61 does not exist, that is, the semiconductor memory 3106 consists of the data memory section 3108 only. In the example, when a flash memory write error occurs, memory block use information is retrieved in the memory block management table 3112 to find an empty block in the data memory section 3108, and data is written into the found empty block as an alternate block. Further, block information of the block of the data memory section 3108 used as the alternate block is written into the error information register 3105 to inform the host system 3101. Other operation of the semiconductor disk system shown in FIG. 75 is the same as that shown in FIG. 61.

Detailed Description Text (125):
FIG. 82 is a block diagram of an information processing system according to the fourth embodiment of the invention, wherein numeral 4001 is a CPU (central processing unit) which executes programs and processes data, numeral 4002 is a flash memory which is a large-capacity nonvolatile memory storing the programs, data, etc., handled by the CPU 4001, and numeral 4003 is a cache memory which is a volatile memory temporarily storing data such as data transferred from the flash memory and write data from the CPU 4001. The cache memory 4003 can be made of a DRAM (dynamic random access memory), an SRAM (static random access memory), or the like, for example. Numeral 4004 is an address array for recording CPU addresses assigned to data stored in the cache memory 4003, which are output by the CPU to access the data, and their appendant information. Numeral 4005 is an address comparison circuit for comparing the address corresponding to the data whose access is requested by the CPU 4001 with the addresses recorded in the address array 4004. Numeral 4006 is a controller which controls the information processing system so that accurate data can be accessed at high speed as much as possible in response to an access request made by the CPU 4001. Numeral 4007 is a memory bus of the CPU 4001. The address array 4004, the address comparison circuit 4005, and the controller 4006 can be made of electronic devices such as CMOS (complementary metal oxide semiconductor) transistors, resistors, ROM which stores predetermined processing programs, RAM, and CPU. In the description that follows, assume that the address space in which addresses accessed by the CPU 4001 exist is allocated to the flash memory 4002. Numeral 4060 is input means having at least a function for an external system including the user to enter commands such as data access commands and addresses; for example, it is implemented by a mouse or keyboard. Numeral 4061 is means having an output function of messages (described below) and necessary information; it can be implemented by print means such as a printer or display means such as a CRT, EL display, or liquid crystal display.

Detailed Description Text (129):
Referring again to FIG. 82, the CPU 4001 accesses the cache memory 4003 via the memory bus 4007. The access address is input to the address comparison circuit 4005, which then compares the address with addresses previously registered in the address array 4004. If the address matches one of the registered addresses, which will be hereinafter referred to as an "address hit," the controller 4006 accesses the location in the cache memory 4003 corresponding to the address. In contrast, if the address does not match any of the registered addresses, which will be hereinafter referred to as an "address miss," the controller 4006 registers the address in the address array 4004. After this, the controller 4006 transfers the data corresponding to the address to the cache memory for storage and accesses the location in the

flash memory 4002 corresponding to the address.

Detailed Description Text (131):
In FIG. 83, numeral 4011 is a DMA (direct memory access) controller for generating consecutive addresses at high speed and accessing the memories at high speed; it is used when consecutive data of several ten to several hundred bytes is transferred. Numeral 4012 is a memory control signal generation circuit for generating control signals to control the operation of the cache memory, address array, etc., as well as the flash memory. For example, it can be implemented by CPU, ROM, RAM, gates, or a program stored in ROM. If DRAM is used as the cache memory, etc., the memory control signal generation circuit may provide a refresh controller to refresh the DRAM. Numeral 4013 is a memory control timer which is means for measuring the time of erasure, write, etc., of the contents of the flash memory; it can be implemented by CPU, ROM, RAM, CMOS, or a program stored in ROM. Numeral 4014 is a volatile data memory used as a work area for control program execution or an area for temporarily storing data from the memory bus, address array information, or data transferred in the system. Numeral 4015 is a ROM which stores a control program and numeral 4016 is a processor which executes the control program for controlling the entire information processing system of the invention. Numeral 4017 is a bus provided to transfer addresses, data, etc., within the system.

Detailed Description Text (133):
As shown in FIG. 84, first an access request to one address is issued from the CPU 4001 at step a. The address is input to the address comparison circuit 4005, which then compares the address with the addresses registered in the address array 4004 at step b. If an address hit occurs, it means that desired data exists in the cache memory 4003. Then, the address is converted into its corresponding address in the cache memory 4003 in response to the information stored in the address array 4004 at step c, and the corresponding location of the cache memory 4003 is accessed according to the cache memory address at step d. That is, if a read access is made, the data in the cache memory 4003 is output to the CPU 4001 via the memory bus 4007; if a write access is made, data input from the CPU 4001 via the memory bus 4007 is written into the corresponding location in the cache memory 4003. On the other hand, if an address miss occurs, a new data area for storing data is created in the cache memory 4003 at step e. Process at step e will be discussed in detail following the description of the flowchart.

Detailed Description Text (137):
Now, if the information processing system does not perform much processing after it starts operation, the cache memory contains a large number of unused data blocks and a new storage area may be allocated in the unused data area without performing a special process. However, if the unused area is not available, a data storage area must be created even by erasing already stored data. Thus, "access history", which is information indicating in what sequence the CPU has accessed the cache memory, is stored corresponding to each address registered in the address array 4004. Data stored in the cache memory and estimated to be least accessed in the future is found for improving performance such as the percentage of presence of data to be accessed in the cache memory, which will be hereinafter referred to as "hitting average," and the access speed.

Detailed Description Text (138):
For this purpose, information indicating relative oldness of addresses last accessed may be recorded An address, the last access to which is the oldest (existing in the most past direction on the time axis) among the recorded addresses, can be found in response to the record contents, and the data at the address may be erased from the cache memory 4003. If the information concerning the access history is stored in the address array 4004, it is efficient in improvement of the processing speed, etc.

Detailed Description Text (141):
By the way, some problems arise on practical application to adopt the set associative method. When a request to write into one address is received from the CPU 4001 and an address miss occurs, address data in the cache memory is written back into the flash memory and a new data storage area is created, as described above. In this case, flash memory data rewrite takes a long time, leading to extensive lowering of system performance such as lowering the processing speed.

Then, control of the controller 4006 can be executed as shown in FIG. 85 for preventing system performance from being lowered.

Detailed Description Text (156):
The operation is described with reference to FIGS. 86, 87, and 89. First, an access request is received from the CPU 4001 at step a. If an address hit occurs, the cache memory 4003 is accessed at step b; if an address miss occurs, a new storage area to store data is created in the cache memory 4003 at step c.

Detailed Description Text (161):
Another example of using a flash memory chip containing a serial buffer where the serial access start address can be set is discussed. Particularly, first, to access a desired line, the address of the line is input. The intra-line location of the first data to be output when a serial clock is input after data on the line is transferred to the serial buffer can be specified by inputting an address.

Detailed Description Text (171):
First, an access request is received from the CPU 4001 at step a. Next, the address value is compared with the addresses in the address array 4004 to determine an address hit or miss at step b. If an address hit occurs, the cache memory 4003 is accessed. If a miss occurs, one line containing the access address is transferred from the flash memory array 4032 to the serial buffer 4033 at step c. If the access is a read, the top address in the access data line is set at step d. Serial clock 4035 is input to the serial buffer 4033 and desired data is taken out at step e. Next, the data is stored in a new storage area provided in the cache memory 4003 at step f. The CPU 4001 accesses the cache memory 4003 and the accessed data is output onto the memory bus 4007 at step g.

Detailed Description Text (189):
In FIG. 93, numeral 4101 is a CPU, numeral 4102 is a bus, numeral 4300 is a cache memory system in a copy back system, numeral 4104 is a main memory consisting of flash memory chips where one block consists of m bytes (m being an integer), and numeral 4105 is a control circuit. The cache memory system 4300 contains an address array 4310 for retaining address information, a cache memory 4320 for retaining data, and an address comparator 4330 for comparing the addresses in the address array 4310 with an address from the CPU 4101. The cache memory for retaining data, 4320, consists of n m-byte registers 4321 (n being an integer). The address array for retaining addresses, 4310, is made up of n registers 4311 each consisting of an address field a for retaining address information, an erasure information field b for retaining information indicating whether or not the block in the memory 4104 corresponding to the address information retained in the address field a is already erased, and an update information field c for retaining information indicating whether or not the corresponding register in the address array 4310 is updated. The update information field c is an already existing field, but the erasure information field b is a new field provided in the invention. Numerals 4312 and 4322 are control signals of the address array 4310 and the cache memory 4320.

Detailed Description Text (190):
FIG. 94 is a process flowchart of the control circuit 4105 in FIG. 93. The example system in FIGS. 93 and 94 assumes that the CPU 4101 always accesses the memory 4104 in block size units of the memory 4104.

Detailed Description Text (202):
FIG. 95 shows a schematic block diagram of an example system when the size of access data to the memory 4104 from the CPU 4101 is smaller than the block size of the memory 4104. Members identical with those shown in FIG. 93 are denoted by the same reference numerals in FIG. 95.

Detailed Description Text (204):
FIG. 96 shows a process flow of the control circuit 4105 in FIG. 95. The process flow is discussed. In the process, for the block erasure operation in the example system in FIG. 93, steps taken for the difference between the access data size and the block size is only described. A process flow considering the block erasure operation is described below.

Detailed Description Text (209):
According to the example system discussed with reference to FIG. 95, even if the size of a write access from the CPU 4101 is smaller than the block size of the memory 4104, data in the block to be written back is temporarily stored in a register 4340 and only the block part into which new data is to be written is updated in the register 4340, then all the data in the register 4340 is written into the corresponding block of the memory 4104 in a batch, thereby enabling a partial write in the block. Even if the size of a read access from the CPU 4101 is smaller than the block size of the memory 4104, all data in a given block is temporarily stored in a register 4321 and only the corresponding data in the register 4321 is output to the bus 4102, whereby partial data in the block can be read out.

Detailed Description Text (211):
Steps identical with or similar to those previously described with reference to FIGS. 94 and 96 are denoted by the same reference numerals in FIG. 100. Step 223 in FIG. 96 is replaced with step 801 and steps 322 in FIG. 94 and step 313 in FIG. 96 are replaced with step 802. Step 803 is newly added preceding step 33 in FIG. 94. The flow enables previous block erasure if the size of a write access from the CPU 4101 is smaller than the block size of the memory 4104.

Detailed Description Text (212):
The flow in which a cache memory hit occurs at step 3 and the information in the erasure information field b of the corresponding register 4313 indicates that the block of the memory 4104 corresponding to the memory write access is not updated and thus control goes to step 803 is possible under either of the following two conditions: In one condition, before update is executed by the hit memory write access, steps 224 and 225 are executed by a miss read access, thereby reading data from block of the memory 4104 and storing the data in register 4321. In this case, the read block of the memory 4104 is not erased. Thus, if a write access is made to the same address in the state, it becomes a cache hit memory write access and control goes to step 803 from step 3. In the other condition, a memory write access to the memory 4104 occurs when only the data in the memory 4104 is valid and the data in the cache memory registers 4321, 4313, 4340, 4350 is invalid, namely, in the empty state after the off-to-on transition of power is made. Since register replacement does not occur at this time, the memory write access can be handled as a cache hit. The corresponding block in the memory 4104 is only erased and update is omitted. Only the corresponding register 4321 is updated. Therefore, control goes to step 803 from step 3.

Detailed Description Text (227):
Data in the cache memory or address array is written into the flash memory periodically, whereby the data will not be lost, or the amount of lost data can be reduced, even if the power supply is stopped abruptly.

WEST

End of Result Set

☐ Generate Collection Print

L91: Entry 1 of 1 File: USPT Jan 9, 2001

DOCUMENT-IDENTIFIER: US 6173385 B1
TITLE: Address generator for solid state disk drive

US Patent No. (1):
6173385

CLAIMS:

1. A method of generating an address for addressing a memory array, comprising:

receiving from a computer bus interface port a block number and a programmable block size value;

providing a multiplier circuit;

changing the block size value in response to a mode command;

multiplying the block number by the block size value in the multiplier circuit; and

providing the product of the step of multiplying from the multiplier circuit to the memory array as an address.

WEST

Generate Collection Print

L53: Entry 3 of 4 File: USPT Jan 9, 2001

DOCUMENT-IDENTIFIER: US 6173385 B1
TITLE: Address generator for solid state disk drive

Abstract Text (1):
An address generator for a solid state disk drive device includes a hardware multiplier logic circuit dedicated to computation of the address by multiplying a lock size by a logical block number, to obtain the start address for a memory array read or write operation. The dedicated multiplier circuit advantageously provides very quick computation of these relatively large numbers, which typically involves a 32 bit by 16 bit multiplication. The multiplier includes a shift register initially holding the logical block number which is shifted a particular number of times, the number of shift pulses representing a value of the block length. The output of the shift register is the desired address.

Brief Summary Text (7):
Referring to the system diagram in FIG. 1, the SCSI host computer 10 presents the SSD 12 with a command to read or write along with the desired logical block and the number of blocks to be written or read on SCSI bus 14. The SSD controller 16 must interpret this command received from SCSI interface 18 and set the DRAM memory array 22 (DRAM chips) to the appropriate starting address via address generator 26. This is analogous to the seek time in a rotating (physical) disk drive. The controller 16 then reads or writes the proper number of blocks to the memory array 22 on address bus 24. (Block-length oriented SSDs are known in the art.) Address generator 26 operates in response to signal AGEN from controller 16.

Brief Summary Text (9):
A difficulty arises due to the variable length of the logical blocks. Consider a typical host computer system 10 based on a logical block length of 512 bytes per block. Block zero begins at an address of zero, block one begins 512 bytes later, block two at 1024 bytes and so on. Should a second system be based upon a block size of 256 bytes per block these addresses would change. In this case block zero would still begin at byte zero, but block one would occur at byte 256, block two at byte 512 and so on.

Brief Summary Text (13):
The present invention in one embodiment includes a dedicated logic circuit for determining the desired starting address of an SSD operation (read or write). The circuit is a multiplier circuit which, as data block numbers are received by the multiplier, multiplies the data block number by a block size value to quickly generate the address. The block size value is (in one embodiment) programmed once, at power up of the SSD, with the appropriate value for the desired block size. (Alternatively the block size value is dynamically alterable after power up). The access time is in the submicrosecond range for any typical block size.

Brief Summary Paragraph Equation (1):
Starting Address=Desired Logical Block.times.Logical Block Length

Detailed Description Text (2):
A SCSI peripheral is a block oriented device. Read or write operations specify a starting logical block, and the number of blocks to be written or read. The logical block size is selected by the host computer using a mode select command. The SCSI peripheral stores the logical block size in a non-volatile location. This logical block size remains valid until a subsequent mode select command modifies the block

size again.

Detailed Description Text (3):
During the initial power up phase, a SCSI SSD retrieves the appropriate size value for logical blocks. The correct logical block size multiplication factor is then programmed into the address generation circuit in accordance with the invention.

Detailed Description Text (5):
The microprocessor 16 then reads the command packet from the SCSI interface 18 buffer. If the command is determined to require reading or writing data, then the microprocessor 16 transfers the starting logical block information directly to the address generation circuit 26.

Detailed Description Text (7):
Separately, at power up of the address generation circuit 26, a block size value is provided on an 8 bit bus 62 as an input data signal to a registered counter 66 which is an 8 bit device. (In this case the input data signal is an 8 bit multiplication factor, not the actual block size, but derived therefrom as described below.) The registered counter 66 has a "load" terminal which responds to the AGEN* (address generation) signal transmitted from microprocessor 16 of FIG. 1. Upon receipt of the AGEN* signal, the registered counter 66 transmits the block size data value held in its register to a NAND gate 70. The second input to NAND gate 70 is a DCLK signal which is the system MEMORY clock signal. Thus, NAND gate 70 triggers the shift register 58 to engage in shifts a number of times equal to the data value held in the registered counter 66, when signal AGEN* is received. The output of the registered counter 66 also is a clock signal to the 32 bit counter block 74 as shown via flip-flop 78. The values held in shift register 58 are transferred to the counter 74 which then transmits them out as a address on an address bus 24 to the random access memory array 22 of FIG. 1.

Detailed Description Text (10):
The address data from the latches U37, . . . ,U40 is loaded into the shift register string U18, . . . ,U24. Then every clock signal supplied to the shift registers U18, . . . ,U24 on their CLK terminals multiplies the address data by two. Thus for large block sizes the shift register-multiplier U18, . . . ,U24 will be clocked proportionally more times than for small block sizes.

Detailed Description Text (14):
This multiplication process continues until the internal counter of counter U44 "rolls over" and sets its output signal ripple carry RCO low. This low signal propagates through flip-flop U42A and stops the clock signal to the shift register-multiplier U18, . . . ,U24. Finally this low signal propagates through flip-flop U42B and loads the multiplied address data (i.e., the start address) into the counter string consisting of counters U6, U7, U8, U9, U10 and U11 by clocking each of these counters. The 24 bit output signal of counters U6, . . . ,U11 are provided to the memory array as the starting address for reading or writing this particular logical block. (Only 24 address bits are needed in this particular embodiment.) Output buffers U1, U2, U3 in turn drive resistor blocks RP1, RP2, RP3 which drive address lines AL0, . . . , AL10, AH0, . . . , AH10 and BS0, BS1. Buffers U1, U2, and U3 are required to drive the address lines on as many as 16 memory modules.

Detailed Description Text (18):
The multiplication factor (logical block size value) is adjusted due to the nature of the registered counter U44 used in this circuit. As previously mentioned, the registered counter U44 is an 8 bit device. This implies a count range from 0 to 255 with terminal carry at count 255. Each clock signal of the shift register is a multiply by two, so that one clock=2.times., two clocks =4.times., three clocks =8.times. etc. The correct preset value for the register part of register-counter U44 is therefore 255 minus the desired number of clocks.

Detailed Description Text (19):
Naively, it would seem that the desired multiplication factor would be directly the logical block size. Under these conditions, because 512 equals 2.sup.9, the correct value for the register would be 255-9=246. This is correct if the memory array

produces an output that is only one byte wide. However solid state disks can be implemented with a memory array that is considerably wider if single byte resolution is not necessary.

Detailed Description Text (20):
An exemplary solid state disk has a memory bus that is eight bytes wide. Eight equals 2.sup.3, and therefore the correct block size value factor for this system would be 255-9+3=249, assuming a 512 byte logical block. This is implemented by a lookup table in the microprocessor 16. The multiplier is the required number of clocks to be presented to the shift register, i.e., the multiplication factor. The circuit stops the multiplication process at rollover, thus the "register value" =255-multiplier. The "register value" is the actual value supplied to registered counter U44.

Detailed Description Text (22):
It is to be understood that the present invention, in addition to being compatible with the SCSI interface, is also compatible with other computer bus peripheral interfaces such as IPI or DSSI or similar interfaces that are block specific. More generally, the present invention is applicable to any interface for computer systems which address memory using a logical block number and a block size, and in particular where the block size is of varying (programmable) length.

Detailed Description Paragraph Table (1):
Logical Block Size Multiplier Register Value 256 5 250 512 6 249 1024 7 248 2048 8 247 4096 9 246

CLAIMS:

1. A method of generating an address for addressing a memory array, comprising:

receiving from a computer bus interface port a block number and a programmable block size value;

providing a multiplier circuit;

changing the block size value in response to a mode command;

multiplying the block number by the block size value in the multiplier circuit; and

providing the product of the step of multiplying from the multiplier circuit to the memory array as an address.

WEST

Generate Collection Print

L30: Entry 1 of 8 File: USPT Oct 31, 2000

DOCUMENT-IDENTIFIER: US 6141728 A
TITLE: Embedded cache manager

Brief Summary Text (15):
The present invention provides efficient and quick scan of the cache list to find every data block requested and transfer the data if necessary. It saves the cache manager from the labor intensive task of searching for data, which reduces performance of the cache manager. Given a starting data block in a requested set, unlike existing systems, the present invention searches to find data blocks beyond the requested data blocks. As a result, after traversing the cache list, the cache manager can be provided with cache status based on the requested set, including a miss, partial hit, full hit, and the first missing data block which needs to be read into the cache buffer from the disk drive.

Detailed Description Text (137):
A path 583 sends end of cache segment information from the cache-buffer interface 586 to the buffer manager 564. A path 585 sends a buffer load, buffer reload control signal from the cache-buffer interface 586 to the buffer manager 564. A path 589 sends a cache segment starting address to the address generator 566 of the cache buffer manager 564. A path 591 returns a buffer address write pointer address from the buffer manager 564 to the cache-buffer interface block 586.

Detailed Description Text (170):
In the FIG. 19 transfer mode, the first hit state 621 is reached following state 648, and state 621 is then followed by five action states: ACT1 656, ACT2 658, ACT3 660, ACT4 662 and ACT5 664. These states represent progressive parallel data calculations and manipulations of the ALU 582 needed to calculate the starting address location value, a trigger count value (number of sectors available in a cache segment for transfer out before a next scan is required) and a segment rollover count value, and to load these values into the cache buffer interface block 586. The actual calculations and manipulations of the ALU 582 occurring for states 632, 646, 648, 621, 656, 658, 660, and 662 are set forth for each of the add, compare, negate and equivalence functions of the ALU in FIG. 20.

Detailed Description Text (197):
With reference to FIG. 25 TRIMLO, a write block sequence 750 from the host 5 has a starting address pointer (WS) and an ending address pointer (WE). An existing cache entry sequence 752 includes a starting address pointer (ES) and an ending address pointer (EE). In the trim-low (TRIMLO) example of FIG. 26, the write sequence 750 overlaps in part the cache sequence 752; however, the starting address WS of the write block sequence 750 is lower than the starting address ES of the cache sequence 752. Further, the ending address WE of the write sequence 750 is lower than the ending address EE of the cache sequence 752. An overlap area is designated X.sub.LO, and it is the function of the cache controller state machine 580 automatically to overwrite the X.sub.LO overlap with so much of the write sequence 750 as overlaps with the cache entry sequence 752. The trim-low equations performed by the ALU unit 582 are shown to the right of the trim-low example of FIG. 25.

WEST

Generate Collection Print

L53: Entry 1 of 4 File: USPT Jul 8, 2003

DOCUMENT-IDENTIFIER: US 6591414 B2
TITLE: Binary program conversion apparatus, binary program conversion method and program recording medium

Abstract Text (1):
A binary program conversion apparatus capable of converting an original binary program into a new binary program which runs at higher speed in a target computer having a cache memory. The binary program conversion apparatus comprises an executing part, a generating part and a producing part. The executing part executes the original binary program. The generating part generates executed blocks information indicating first instruction blocks which are executed by the executing part. The producing part produces, based on the executed blocks information generated by the generating part, the new binary program which contains second instruction blocks corresponding to the plural of the first instruction blocks and which causes, when being executed in the computer, the computer to store second instruction blocks corresponding to the first instruction blocks executed by the executing part at different locations of the cache memory.

Brief Summary Text (20):
The generating part generates executed blocks information indicating first instruction blocks which are executed by the executing part.

Brief Summary Text (21):
The producing part produces, based on the executed blocks information generated by the generating part, the second binary program which contains second instruction blocks corresponding to the plural of the first instruction blocks and which causes, when being executed in the computer, the computer to store second instruction blocks corresponding to the first instruction blocks executed by the executing part at different locations of the cache memory.

Brief Summary Text (25):
The controlling part controls the executing part so as to execute a third binary program which consists of a plural of third instruction blocks at first. Then, the controlling part controls the generating part so as to generate second executed blocks information indicating third instruction blocks executed by the executing part. Moreover, the controlling part controls the producing part so as to produce a fourth binary program which causes, when being executed in the computer, the computer to store fourth instruction blocks corresponding to the third instruction blocks executed by the executing part on different locations on lines of the cache memory excluding the lines indicated by the line data.

Brief Summary Text (37):
The binary program conversion method according to the first aspect comprises an executing step, a generating step and a producing step. In the executing step, the first binary program is executed. In the generating step, executed blocks information indicating first instruction blocks executed in the executing step is generated. In the producing step, based on the executed blocks information generated in the generating step, the second binary program is generated which contains second instruction blocks corresponding to the plural of the first instruction blocks of the first binary program and which causes, when being executed in the computer, the computer to store the second instruction blocks corresponding to the first instruction blocks executed in the executing step on different locations of the cache memory.

Brief Summary Text (39):
It is feasible to further add, to the program conversion method, a creating step and a controlling step. The creating step involves a process of creating a line data indicating lines of the cache memory which are to be used when the second binary program produced in the producing step is executed in the computer. The controlling step involves processes of controlling the executing step so as to execute a third binary program which consists of a plural of third instruction blocks, and of controlling the generating step so as to generate second executed blocks information indicating third instruction blocks executed in the executing step, and of controlling the producing step so as to produce, based on the second executed blocks information and the line data, a fourth binary program which contains fourth instruction blocks corresponding to the plural of the third instruction blocks and which causes, when being executed in the computer, the computer to store the fourth instruction blocks corresponding to the third instruction blocks executed by said executing means at different locations on lines of the cache memory excluding the lines indicated by the line data.

Detailed Description Text (45):
Next, the object code converter determines the instruction block Sj which will be copied into the CaB area next, and computes MADDR(Sj) which is the starting address of the instruction block Sj (step 123). Then, the object code converter judges whether a branch instruction Ci is included in the block Si (step 124).

Detailed Description Text (64):
When starting the address constant relocation process, the object code converter selects an address constant among the address constants which are stored in the executable object program by the link editor, and judges whether the selected address constant points to any locations on the instruction blocks whose copies are made on the CaB (step 170). When the address constant points to any of such locations (step 170:T), the object code converter rewrites the address constant with a corresponding address in the CaB area (step 171) and proceeds to step 172. On the contrary, when the address constant does not point to any of such locations (step 170:F), the object code converter proceeds to step 172 without rewriting the address constant.

Detailed Description Text (71):
To begin with, the basic operation procedure of the binary conversion apparatus will be discussed by exemplifying a case where one executable object program whose list is shown in FIG. 19 is converted. Note that, the instruction "seticc" in the list represents an instruction for setting a condition code to which conditional branch instruction refers. The instruction "be" (branch equal) is an instruction which causes a branch when an equal condition is satisfied. The "c-." following "be" indicates that the offset value of the branch target is a value obtained by subtracting the present address (".") from the starting address of the block c. The instruction "delay" is a delay instruction which is executed at the same time with the branch instruction. The "delay" is executed even when the branch is taken. That is, after a branch instruction and corresponding "delay" are executed, an instruction of the branch target is executed.

Detailed Description Text (74):
That is, the unconditional branch instruction "ba, a a'-." which causes a branch to the block a' in the CaB area is set at the starting address of the block a in the MTXT area (*1). Similarly, the unconditional branch instruction "ba, a b'-." which causes a branch to the block b' is set at the starting address of the block b (*2). Note that, ", a" in the instruction "ba, a a'-." or "ba, a b'-." represents that the instruction is an annulled branch instruction which cause the CPU not to execute its succeeding instruction.

Detailed Description Text (77):
In this case, the program shown in FIG. 19 is converted into the program shown in FIG. 22. That is, the unconditional branch instructions to the blocks a' and c' are set at the starting addresses in the blocks a and c in the MTXT area (*5,*7). Moreover, since the branch target of the branch instruction "be" in the block a is the instruction block c whose copy is made on the CaB area, the target offset value

of the branch instruction is rewritten so that the branch instruction causes a branch to the instruction block c' (*6).

CLAIMS:

1. A binary program conversion apparatus, used for converting a first binary program which consists of a plurality of first instruction blocks into a second binary program which is executed in a computer having a cache memory, comprising: executing means for executing the first instruction blocks of the first binary program after an additional area has been added to the first binary program, the additional area having a size of up to and including a size of the cache memory; generating means for generating executed blocks information from the executed first instruction blocks; and producing means for producing the second binary program from the executed blocks information, the second binary program comprising second instruction blocks corresponding to the first instruction blocks, ones of the second instruction blocks having been copied into the addition area wherein when the second binary program is executed in the computer, the computer stores the second instruction blocks corresponding to the executed first instruction blocks at locations where any other second instruction blocks are not stored within the cache memory, and object code of the first binary program is used during a process of converting the second binary program from the first binary program.

3. A binary program conversion apparatus according to claim 1, further comprising: creating means for creating a line data indicating lines of the cache memory which are to be used when the produced second binary program is executed in the computer; and controlling means for controlling said executing means so as to execute third instruction blocks of a third binary program, said generating means so as to generate second executed blocks information from the executed third instruction blocks, and said producing means so as to produce a fourth binary program from the second executed blocks information and the line data, the fourth binary program comprising fourth instruction blocks corresponding to the executed third instruction blocks, wherein when the fourth binary program is executed in the computer, the computer stores the fourth instruction blocks corresponding to the executed third instruction blocks at locations in lines of the cache memory except the lines indicated by the line data.

10. A binary program conversion method, used for converting a first binary program having first instruction blocks into a second binary program which is executed in a computer having a cache memory, comprising: executing the first instruction blocks of the first binary program after an additional area has been added to the first binary program, the additional area having a size of up to and including a size of the cache memory; generating executed blocks information from the executed first instruction blocks; and producing the second binary program from the executed blocks information, the second binary program comprising second instruction blocks corresponding to the first instruction blocks of the first binary program, ones of the second instruction blocks having been copied into the additional area, wherein when the second binary program is executed in the computer, the computer stores the second instruction blocks corresponding to the executed first instruction blocks at locations where any other second instruction blocks are not stored within the cache memory, and object code of the first binary program is used during the process of converting the second binary program from the first binary program.

12. A binary program conversion method according to claim 10, further comprising: creating a line data indicating lines of the cache memory which are to be used when the produced second binary program is executed in the computer; and controlling said executing executes a third binary program comprising third instruction blocks, and said generating generates second executed blocks information from the executed third instruction blocks, and said producing produces a fourth binary program from the second executed blocks information and the line data, the fourth binary program comprising fourth instruction blocks corresponding to the executed third instruction blocks, wherein when the fourth binary program is executed in the computer, the computer stores the fourth instruction blocks corresponding to the executed third instruction blocks at locations on lines of the cache memory except the lines indicated by the line data.

16. A recording medium for recording programs for making a computer having a cache memory function as: executing means for executing first instruction blocks of a first binary program after an additional area has been added to the first binary program, the additional area having a size of up to and including a size of the cache memory; generating means for generating executed blocks information from the executed first instruction blocks; and producing means for producing a second binary program from the executed blocks information, the second binary program comprising second instruction blocks corresponding to the executed first instruction blocks of the first binary program, ones of the second instruction blocks having been copied into the additional area, wherein when the second binary program is executed in the computer, the computer stores the second instruction blocks corresponding to the executed first instruction blocks at locations where any other second instructions blocks are not stored within the cache memory, and object code of the first binary program is used during the process of converting the second binary program from the first binary program.

17. A binary program conversion apparatus, used for converting a first binary program including a plurality of object modules having a plurality of first instruction blocks into a second binary program which is executed in a computer having a cache memory without using a source code of the first binary program, comprising: adding means for adding an additional area to the first binary program, the additional area having a size up to and including a size of the cache memory, and adding instructions of the additional area being jumped to from the first instruction blocks; executing means for executing first instruction blocks of the first binary program with the additional area added; generating means for generating executed blocks information from the executed first instruction blocks; and producing means for producing the second binary program by copying the first instruction blocks of the plurality of object modules into different areas where any other second instruction blocks of the second binary program are not stored within the additional area based on the executed blocks information, the second binary program comprising the second instruction blocks corresponding to the first instruction blocks, wherein when the second binary program is executed in the computer, the computer stores the second instruction blocks corresponding to the executed first instruction blocks at locations where any other second instruction blocks are not stored within the cache memory, and object code of the first binary program is used during a process of converting the second binary program from the first binary program.

19. A method of converting a first binary program comprising a plurality of object modules having first instruction blocks into a second binary program which is executed in a computer having a cache memory without using a source code of the first binary program, comprising: adding an additional area to the first binary program, the additional area being an additional area having a size corresponding to and including a size of the cache memory, and adding instructions of the additional area being jumped to from the first instruction blocks; executing the first instruction blocks of the first binary program; generating executed blocks information from the executed first instruction blocks; and producing the second binary program by copying the executed first instruction blocks of the plurality of the object modules into different areas within the additional area based on the executed blocks information, the second binary program comprising second instruction blocks corresponding to the executed first instruction blocks, wherein when the second binary program is executed in the computer, the computer stores the second instruction blocks corresponding to the executed first instruction blocks at locations where any other second instructions blocks are not stored within the cache memory, and object code of the first binary program is used during a process of converting the second binary program from the first binary program.

21. A computer program embodied on a computer-readable medium for converting a first binary program comprising a plurality of object modules having first instruction blocks into a second binary program which is executed in a computer having a cache memory without using a source code of the first binary program, comprising: an adding element to add an additional area to the first binary program, the additional area being an additional area having a size up to and including a size of the cache memory, and adding instructions of the additional area being jumped to from the first instruction blocks; an executing element to execute the first instruction

blocks of the first binary program; an generating element to generate executed blocks information from the executed first instruction blocks; and a producing element to produce the second binary program by copying the executed first instruction blocks of the plurality of the object modules into different areas within the additional area based on the executed blocks information, the second binary program comprising second instruction blocks corresponding to the executed first instruction blocks, wherein when the second binary program is executed in the computer, the computer stores second instruction blocks corresponding to the executed first instruction blocks at locations where any other second instruction blocks are not stored within the cache memory, and object code of the first binary program is used during a process of converting the second binary program from the first binary program.

23. A binary program conversion apparatus, used for converting a first binary program comprising first instruction blocks into a second binary program which is executed in a computer having a cache memory without using a source code of the first binary program, comprising: adding means for adding a cache blocking area of predetermined size comprising a plurality of lines to the first binary program; executing means for executing the first binary program; generating means for generating executed blocks of information indicating first instruction blocks in the first binary program executed by said executing means; and producing means for producing, by copying the first instruction blocks into different areas within the cache blocking area based on the executed blocks of information generated by said generating means, the second binary program which contains second instruction blocks corresponding to the plurality of the first instruction blocks and which causes, when being executed in the computer, the computer to store second instruction blocks corresponding to the first instruction blocks executed by said executing means at different locations of the cache memory, wherein the source code of the first binary program is not used to create the second binary program, wherein the producing means compares a total size of the executed first instruction blocks with the size of the cache blocking area, and if the total size of the first instruction blocks is not larger than the size of the cache blocking area, determines a number of the lines, L, necessary for holding all of the executed instruction blocks, selects L number of lines of the cache memory, of which a frequency in use is low, produces the second binary program by copying the first instruction blocks into the location in the cache blocking area, the location corresponding to the selected lines, if the total size of the first instruction blocks is larger than the size of the cache blocking area, selects the larger number of the first instruction blocks in the order of frequency of being executed so that the total size of the selected first instruction blocks does not exceed the size of the cache blocking area, and produces the second binary program by copying the selected first instruction blocks into the cache blocking area.

WEST

Generate Collection Print

L64: Entry 3 of 28 File: USPT Oct 3, 2000

DOCUMENT-IDENTIFIER: US 6128307 A
TITLE: Programmable data flow processor for performing data transfers

Abstract Text (1):
The present invention comprises an architecture that involves an embedded Digital Signal Processor (DSP), a DSP interface and memory architecture, a micro-controller interface, a DSP operating system (OS), a data flow model, and an interface for hardware blocks. The design allows software to control much of the configuration of the architecture while using hardware to provide efficient data flow, signal processing, and memory access. In devices with embedded DSPs, memory access is often the bottleneck and is tightly coupled to the efficiency of the design. The platform architecture involves a method that allows the sharing of the DSP memory with other custom hardware blocks or the micro-controller. The DSP can operate at full millions-of-instructions-per-second (MIPS) while another function is transferring data to and from memory. This allows for an efficient use of the memory and for a partitioning of the DSP tasks between software and hardware.

Brief Summary Text (13):
The configurable data transfer architecture communications system comprises a data flow processor (DFP), a micro-controller unit (MCU) coupled to the DFP, a digital signal processor (DSP) coupled to the DFP, and a memory coupled to the DFP and to the DSP. The MCU controls operations in the communications system, the DSP performs digital signal processing functions and executive functions in the communications system, and the memory stores data used by the DSP and the MCU. The DFP selectively allows

Brief Summary Text (15):
The DFP provides an interface between the MCU, the DSP, and the memory. The MCU and the DSP are operable to communicate with each other and the memory through the DFP which is programmable to perform data transfers between two or more of the MCU, the DSP, and the memory. The MCU is operable to program the DFP to perform data transfers between two or more of the MCU, the DSP, and the memory. In addition, the MCU is operable to transfer a task list to the memory which comprises tasks to be executed by the DSP. The DSP executes the tasks in the task list and the DFP performs data transfers to perform communication functions in the communications system. The MCU programs the DFP to perform data transfers to aid in accomplishing the tasks in the task list.

Brief Summary Text (16):
The memory is directly accessible by the DFP and the DSP through a memory interface, but the MCU accesses the memory through the DFP. The memory can be accessed by a plurality of transfer paths coupled between the memory interface and one of the memory segments. Each of the transfer paths provides an independent data transfer path between the memory interface and a respective memory segment. The DSP and the DFP are operable to simultaneously access different memory segments through the memory interface. Each of the DSP and the DFP are operable to select different ones of the transfer paths for simultaneously connecting to different memory segments.

Brief Summary Text (18):
The DFP comprises a stream processor and programmable registers coupled to the stream processor. The stream processor manages multiple, simultaneous data transfers between two or more of the MCU, the DSP, and the memory through the DFP. The programmable registers are programmable to control operation of the stream

processor. The MCU is operable to write data to the programmable registers to program the stream processor. The data written to the programmable registers includes information indicating the priority of each of said data streams and the amount of data to be transferred for each of the data streams.

Brief Summary Text (19):
The stream processor is operable to access data comprised in the programmable registers and in response configure data streams between a source and a destination. The stream processor is operable to determine when to execute the data streams. The stream processor includes at least one buffer for temporarily storing data transferred between the source and the destination. The stream processor determines an ordering of transfers based on buffer availability. The DFP operates to perform the data transfers without further intervention from the MCU.

Brief Summary Text (20):
In one embodiment, the communications system further comprises a slave DSP coupled to the DFP. The slave DSP is a slave processor to the DSP and is controlled by the DSP. In another embodiment, the communications system further comprises a second DSP coupled to the DFP. The MCU is operable to transfer a second task list to the second DSP with a second plurality of tasks for the second DSP to perform.

Brief Summary Text (21):
In another embodiment, the communications system further comprises a first memory bus and second memory bus. The first memory bus couples the memory to the DFP and the second memory bus couples the memory to the DS-P.

Brief Summary Text (22):
In another embodiment, the communications system further comprises one or more hardware accelerator units coupled to the DFP, The hardware accelerator units access the memory through the DFP and are operable to perform at least a portion of one or more of the tasks in the task list. The hardware accelerator units may comprise a dedicated input/output unit or a dedicated signal processing unit.

Brief Summary Text (23):
Broadly speaking, the present invention comprises an architecture that involves an embedded Digital Signal Processor (DSP), a DSP interface and memory architecture, a micro-controller interface, a DSP operating system (OS), a data flow model, and an interface for hardware blocks. This design allows software to control much on the configuration of the architecture while using hardware to provide efficient data flow, signal processing, and memory access. In devices with embedded DSPs, memory access is often the bottleneck and is tightly coupled to the efficiency of the design. The platform architecture involves a method that allows the sharing of DSP memory with other custom hardware blocks or the micro-controller. The DSP can operate at full millions-of-instructions-per-second (MIPS) while another function is transferring data to and from memory. This allows for an efficient use of memory and for a partitioning of the DSP tasks between software and hardware.

Brief Summary Text (24):
The architecture involves an operating system and data protocol that allows tasks to be efficiently partitioned and scheduled. The plat form operating system is part software and part hardware with the two parts working together. Managing the data flow is done in hardware by the data flow processor (DFP) to support the software. The operating system, running on the DSP and on the micro-controller unit (MCU) has been designed to allow scheduling to be flexible. Many system implementations have tasks efficiently scheduled in a static manner. This often involves each task beginning at a specific point in time with the total number of clock cycles for each task known. However, when a task changes, many details in firmware or hardware must be reworked to accommodate the change. The tight coupling of events often ripples these changes throughout the system. Dynamic scheduling is at the other end of the spectrum with a high cost for overhead and development. The platform architecture allows programmable data driven scheduling. The list of tasks (algorithms, data movement, and some control functions) can be programmed in software. The software can easily handle changes during development and optimization of resources as the hardware and software implementations become final. The architecture can handle applications that involve distributed, concurrent, and data driven processes.

Brief Summary Text (26):
The platform involves a software and hardware definition to implement a system architecture. The software running on the MCU controls the overall operation. The platform is programmed by the MCU during setup. The setup includes several functions such as providing task scheduling to the DSP executive routine, programming the DFP registers for data flow, and programming the hardware blocks. The executive routine running on the DSP controls the tasks running on the DSP from a task list provided by the MCU. The task list may involve controlling hardware blocks to off-load the DSP. The DFP functions as a key part of the platform and serves as a common hardware interface between blocks. The DFP also functions as the main interface to the DSP and the DSP memory. The DFP also contains programmable control for managing multiple simultaneous data flow. The DFP does not provide control to the platform.

Brief Summary Text (28):
The platform is designed to optimize each of the main functions of the system. The overall control is in the software of the MCU. This control is used to setup the platform and to make changes to the tasks during operation. The setup involves the scheduling of tasks which is done by writing a task list to the DSP executive software. During operation, the MCU can process data and execute higher level protocol tasks while the DSP executive software executes the task list independent from the MCU. The MCU will need to make changes to the task list in the DSP as necessary. Programmable hardware in the DFP is used for data flow and block interface. The DFP block determines how data is transferred, not when or where. The MCU programs the DFP for source and destination addresses for repetitive data flows for all tasks, referred to as data streams. The MCU programs data streams prior to real time operation. The DFP and the DSP executive routine then function in real time to efficiently execute data flow by allowing the platform blocks to operate with a minimum of real time control from the MCU for routine or repetitive functions.

Brief Summary Text (31):
The DFP contains a common interface and protocol for all the hardware accelerators. Hardware blocks can be added or deleted without changing the DFP. The DFP is the bus master to the DSP memory and the hardware accelerator bus. While the hardware blocks are slaves on their bus to simplify the design, they control their own input and output data flow through dedicated request signals.

Drawing Description Text (13):
FIG. 12 shows a block diagram of the data flow processor;

Drawing Description Text (15):
FIG. 14 shows the DFP-DSP interface for IRQ0 & IRQ1;

Drawing Description Text (16):
FIG. 15 shows the DFP-RAM interface with single access R/W;

Drawing Description Text (17):
FIG. 16 shows the DFP-RAM interface with double access R/W;

Drawing Description Text (19):
FIG. 18 shows the HW Accelerator-DFP interface;

Drawing Description Text (20):
FIG. 19 shows the DFP-SFT interface;

Drawing Description Text (26):
FIG. 25 shows a flowchart of the EXEC routine command interface.

Detailed Description Text (5):
Digital Signal Processor (DSP)

Detailed Description Text (6):
In a preferred embodiment, DSP 300 comprises an ADI 2171 DSP. The ADI 2171 DSP 300 and memory bus architecture have been modified to allow sharing of DSP memory 350

with the rest of the system while minimizing the impact on DSP MIPS. DSP 300 does not contain the 2171's Host Interface Port. The interface to DSP 300 memory 350 by platform blocks other than DSP 300 is done by the Data Flow Processor (DFP) 320. DSP 300 contains two software functions that are important to the architecture's operation: the interrupt and the executive routines. The interrupt routine handles the interface to DSP 300 from DFP 320 and passes control data along to the executive routine. The executive code is DSP's 300 operating system that manages all functions outside each of the specific signal processing code blacks. This includes temporary data buffers in DSP 300 memory, sending and receiving commands to the rest of the chip, and execution of the task list as programmed by MCU 310. MCU 310 is assumed to execute protocol and some signal processing.

Detailed Description Text (9):
Data Flow Processor (DFP)

Detailed Description Text (10):
DFP's 320 purpose is to transfer data between all the blocks and to master the buses to hardware blocks 330 and to DSP 300 memory bus. DSP's 300 parallel port interface has been replaced by the ability to share blocks of memory under direct access by DFP 320. Therefore, one of the main functions of DFP 320 is to interface with DSP 300 to transfer all data and commands through access to DSP's 300 shared memory 350. DFP 320 contains hardware dedicated to interfacing with DSP's 300 interrupt routine and for memory access. Data is transferred between other blocks in the platform and DSP's 300 memory via DFP 320. DFP 320 is designed to manage continuous data flow through data streams. Data streams, defined below, allow data transfers to be programmed and occur as the tasks are scheduled with minimum software interrupts. Data streams are programmed by MCU 310.

Detailed Description Text (12):
Timer block 340 is a behavioral model containing a timer and a controller for initiating scheduled events in hardware. The protocol software running on MCU 310 controls the modes and timing information for hardware 330. MCU 310 protocol software does not directly control the timing of events within a frame. MCU 310 software will program timer block 340 to initiate any type of task by providing a count to the timer. Block 340 can be programmed for many tasks at once and will generate a data read cycle by

Detailed Description Text (15):
The platform development will contain a behavioral register and bus model for hardware blocks 330 and 335. These blocks may operate as stand alone functions or may be used to accelerate a DSP software function. In FIG. 3, they are referred to as hardware accelerators (HW ACC). The platform will allow blocks 330 to use DSP's 300 memory as input and output buffers. The shared memory architecture and DFP 320 allow DSP 300 to execute while a hardware accelerator runs using data buffers in DSP memory 350B. Each hardware accelerator has control of the data rate from DSP data memory 350B. How the accelerator uses DSP's 300 data memory is flexible. It can be on a block or word basis keeping the register count low in the accelerator.

Detailed Description Text (16):
The design of hardware blocks 330 is completely flexible. A simple common interface to DFP 320 is used for all hardware blocks. Hardware blocks 330 can contain addressable registers for programming operating modes and functions. Second DSP 360 could be used as slave processor to main DSP 300. A second DSP would require program memory and some amount of data memory for processing.

Detailed Description Text (24):
DSP Interrupt--An interrupt of DSP 300 using the IRQ2,IRQ1, or IRQ0 pins on DSP 300. These can be controlled by DFP 320. The IRQ2 interrupt routine on DSP 300 is used as the interface to the rest of the system and is controlled by DFP 320.

Detailed Description Text (31):
Software/Hardware Interface

Detailed Description Text (32):
There are two software-to-hardware interfaces that need to be defined for data modem

systems where at least a portion of signal processing is performed in software on a DSP. One is micro-controller (MCU 310) to hardware interface (including DSP 300). The second is the interface to DSP 300 software. DSP 300 software functions need to be defined for the interface to the hardware portion of the platform as well as for MCU 310 to DSP 300 interface.

Detailed Description Text (33):
Controller Software to Baseband Hardware Interface

Detailed Description Text (34):
The protocol software running on the controller has to be able to determine the baseband operating modes as well as transfer data between the signal processing functions and the protocol. MCU's 310 software can not control any functions that are timing critical in the baseband hardware. For a remote controller, this is a must due to bus latencies. The interface for the software definition would be essentially the same for MCU 310 on or off chip.

Detailed Description Text (35):
One of the interfaces between both MCU 310 and DSP 300 software and the platform hardware is in implementing data flow via data streams. The concept of data streams has been defined above. DFP 320 hardware manages the data streams that transmit data between the software in MCU 310 and DSP 300. The data streams are determined by MCU 310 software programming the hardware for each source/destination pair of addresses for data flow. Data is then transferred when available by DFP 320. This allows the data flow of the system to be completely determined by software.

Detailed Description Text (38):
MCU's 310 behavioral model in the platform uses a 32 bit bus and operation is not specific to a micro-controller. The hardware interface to MCU 310 is straightforward when on chip. DFP 320 interfaces to the MCU's 310 parallel bus. MCU 310 is the master. MCU 310 can address its own memory, DSP's 300 memory, and registers in the platform hardware using the memory addresses in FIG. 4. MCU's 310 address space uses 19 bits. DFP 320 uses a 16 bit address space. DFP 320 maps MCU 310 ADDRESS[17:2] to DFP 320 ADDRESS[15:0]. The platform hardware registers and DSP's 300 memory are part of the 16 bit DFP space, see FIG. 5. DSP 300 uses a 14 bit address space to address the 16K words of program or 16K words of data memory. The selection of program or data memory is done by DSP's 300 bus interface. The memory map for DSP 300 is given in FIG. 6.

Detailed Description Text (40):
One of the first tasks of the controller is to setup DSP 300 for the task list. MCU 310 will program DFP 320 for a data stream to DSP 300 to be used for commands. MCU 310 will send data commands on each task to this data stream and DFP 320 will write the data into DSP's 300 data memory and interrupt DSP 300. The interrupt routine and executive routines will store the information on each task. Examples of information sent to DSP's 300 executive routine are:

Detailed Description Text (45):
The task lists for the main operating modes will generally be a list stored in memory and MCU 310 will down load the list. During operation, MCU 310 can make changes to the task list be sending commands to change the effected tasks. Each of the tasks running from DSP's 300 EXEC software requires corresponding data streams to be programmed in DFP 320.

Detailed Description Text (47):
DFP 320 is programmable for the specifics of each data stream. MCU 310 must program the data streams by writing to the stream control registers within DFP 320. MCU 310 programs as many data streams as necessary to support the task schedule. Examples of information written to the stream control registers are the source and destination address, the packet size for individual data transfers, and the buffer size. The programming of DFP 320 is done up front by MCU 310. DFP 320 maintains many data streams simultaneously, for the platform model it is 32. For data flowing between two addresses throughout the operation of the system, MCU 310 will not be involved further. As functional modes change, MCU 310 may turn off some data streams in order to use the resources in DFP 320 for others. The data stream needs of the platform

will be determined during the design and MCU 310 will need to manage the operating modes, task schedule, and the data streams that support the programmed tasks. Setup and changes to DFP 320 should be not be a real time function (cycle accurate) for MCU 310.

Detailed Description Text (49):
In step 962, the MCU generates a task list that comprises a plurality of tasks to be executed by the DSP. The task list is transferred to the DFP, in step 964, which transfers the task list to the memory in step 966. The DFP interrupts the DSP to alert the DSP to the presence of the task list in the memory.

Detailed Description Text (50):
In step 968, the DSP accesses the task list from the memory in response to the DFP's interruption, and, in step 970, the DSP executes the task list to perform communication functions. Executing the task list comprises the DSP configuring the DFP to execute one or more data streams, the DSP determining a buffer allocation in the memory, and communicating to the DFP that the buffer in the memory is available or full. In step 972, the DFP performs the data transfers in response to the DSP communicating to the DFP that the buffer in the memory is available or full.

Detailed Description Text (51):
In one embodiment, the DSP distributes at least a portion of one or more of the tasks to one or more auxiliary DSPs through the DFP. To accomplish this, the DSP writes a second task list to the memory, the DFP reads the second task list from the memory, and the DFP transfers the second task list to the one or more auxiliary DSPs. The one or more auxiliary DSPs execute the tasks on the second task list.

Detailed Description Text (53):
The timer block can be used to initiate a transmit or receive algorithm sequence or to time the transfer of data from one function to another. The timer provides a counter triggered output to other blocks in the platform by a direct signal or W/R cycle through DFP 320. MCU 310 programs the hardware timing block by writing the value of the counter to the timer's registers along with an address and data. When the timer count is met, the timer control will initiate a DFP transfer of the address/data. The timer block could be used to frame synchronize MCU 310 and analog functions directly. The timer control can handle many programmed events at once.

Detailed Description Text (54):
DSP Software to Baseband Hardware Interface

Detailed Description Text (55):
There are two functional blocks of DSP code that provide operating system functions. These are the interrupt routine and the executive routine (EXEC). These blocks of code are separate from the signal processing algorithms. The interrupt routine should be kept small, about 100 words. The interrupt routine interfaces with DFP 320 for memory allocation for data buffers, status, and data stream control. The executive routine is the software interface between DSP's 300 code and the rest of the chip. Some of the main functions of DSP's 300 OS software are listed below. The details are given in the part of the description on Software OS Functions.

Detailed Description Text (61):
2.2. The interrupt routine will perform some checks against invalid data stream commands such as referencing a stream number that is not part of the task list.

Detailed Description Text (62):
3. Interface to the EXEC

Detailed Description Text (64):
3.2. The interrupt routine interfaces to the EXEC for passing message pointers, providing data buffers for streams, and updating registers about data streams for the EXEC to process at its leisure.

Detailed Description Text (67):
The executive (EXEC) routine is the operating system that interfaces to and manages the individual signal processing blocks of code. The EXEC should be about 500-1K

words. The EXEC is an operating system that is programmable by MCU 310. It contains the basic information to manage data streams, algorithms, data memory buffers for control commands (messages), and how to interface with the interrupt routine and DFP 320's memory mapped registers. The EXEC is generally programmed by MCU 310 for the data stream and algorithm scheduling before execution and algorithms are initiated from data flow or the hardware timer. MCU 310 can send and receive messages directly to the EXEC using a data stream. For data modem systems, there is often a need to abort an algorithm and retrain. Messages allow a direct communication between MCU 310 and DSP EXEC.

Detailed Description Text (79):
2. Interrupt Routine Interface

Detailed Description Text (85):
3.2. the EXEC manages the order of operations. When executing the task list, the EXEC manages the input buffers for upcoming algorithms and allows reuse of data buffers for completed algorithms.

Detailed Description Text (86):
3.3. scheduled algorithms to execute, according to the conditions provided in the task list. One of the conditions will require an input data buffer to be full. The EXEC will be told by DFP 320 or an algorithm when data buffers are full or empty. An algorithm can begin execution by the EXEC receiving a message from the hardware timer. This provides timed scheduling (within several instruction cycles) and data flow scheduling together.

Detailed Description Text (89):
MCU 310 will download the task list, as mentioned before, to DSP's 300 executive. The execution of the task list schedules more than just the order of DSP's 300 algorithms. The scheduling can also involve data streams, DSP memory buffers, and the timer count. An `example` of how tasks may be scheduled is given below. This represents the functional information that is controllable through the task list.

Detailed Description Text (109):
DSP's 300 executive routine (EXEC) will perform management of the task list using the data streams. The EXEC will run occasionally from the interrupt routine and in between algorithms. The EXEC will perform memory management of data buffers located in DSP memory. The output buffer of a software algorithm could also be used as an input buffer to a hardware accelerator. DFP 320 would read the data and write it to the HW ACC block at the rate needed by the HW ACC block. A software algorithm and a HW ACC could both use the same data buffer as input to co-process two separate tasks.

Detailed Description Text (110):
The task list gives the conditions on which the EXEC should move from task to task. Multiple tasks could be enabled simultaneously in which case each would run when the input and output DSP memory buffers allow. An example would be processing lower rate audio data `as it is available` while processing another task the majority of the time. DSP's 300 software algorithms need to update a status register when they have completely read or written a DSP memory buffer. The tasks are called from the EXEC and return to the EXEC unless they are actively processing. If the task is awaiting data but not complete, the task will be re-called until complete. The tasks in the task list, software and hardware, control the data rates to and from the task. This allows processing efficiencies and changes to be independent of the interface.

Detailed Description Text (111):
When the task list is complete, DSP 300 will enter IDLE mode and reduce power. DSP 300 exits IDLE when interrupted. DFP 320 controls DSP's 300 interrupt and would do so when a data stream buffer has been fully written/read. This provides a power down that is exited automatically due to the data flow, MCU, or timer event.

Detailed Description Text (113):
This part describes each of the functional blocks in the platform. The system level task scheduling and data flow are implemented by the software and hardware execution of data streams. Each of the main blocks has a hardware and software function. The

architecture partitions the control functions into software, configuration of hardware into programmable registers, and the data processing in DSP code or hardware. FIG. 8 shows this partitioning. The data flow processor is hardware to support the data flow and many of the operational details are given in that part. The 21xx DSP is embedded and the pin functions are given below.

Detailed Description Text (118):
The following parts will provide a more detailed description of the interface between DSP 300, data flow processor 320, and the RAMs 350A and 350B. The platform contains the full addressable program and data memory RAM space on chip, 16K each. The 21XX Host Interface Port has not been included. DFP 320 performs the function of interfacing to DSP 300 and RAM 350. The RAM architecture and interface to the core is different from the 2171. The program and data RAM have each been partitioned into 4K blocks. A diagram of the memory architecture is given in FIG. 10. Memory is referenced by a 15 bit address space generated by DFP 320. DFP 320 outputs the bus memory address, BMA[14:0]. The RAM address space is given below where the msb selects the program or data memory and bits [13:12] select the 4K block of memory. The next part describes DFP 320 and includes details on the signal and functional interface to memory 350.

Detailed Description Text (119):
The impact on DSP's 300 code for this memory structure is there is no external memory bus. DSP's 300 operation with the GO bit is similar. The GO bit should always be enabled. Up front scheduling of operations should keep DSP 300 and DFP from the same block of memory for long periods of time. DSP's 300 code has no way of knowing where DFP 320 is accessing. The IRQ2,Flag IN and Flag Out pins are dedicated to the interface to DFP 320.

Detailed Description Text (120):
DFP 320 has direct memory access to service the data streams and interface with DSP 300. DFP 320 and DSP 300 have the ability to each write to two different blocks of memory simultaneously. They can not access the same block at the same time. DFP 320 writes to both the program and data memory at the instruction rate. It takes one instruction cycle to give the RAM bus back to DSP 300 after each DFP RAM access. The instruction immediately following the last DFP read or write can not be used by DSP 300 for an access to the same memory block. DFP 320 uses a modified version of bus request and grant to gain access to the memory buses. A more detailed

Detailed Description Text (121):
diagram of the RAM interface is shown in FIG. 10. This block diagram does not show all signals that interface DSP 300, memory interface and bus interface blocks. DSP 300 provides eight sets of addresses for the data memory 350B and program memory 350A, DM[1-4]A[11:0] and PM[1-4]A[11:0]. These are used to address the 4K address space for e memory block. The selection of the memory block is done by the RD, WR, & PC signals from DSP 300 or by the upper 3 bits of the BMA address. Each RAM block has a data in and data out bus that is muxed & demuxed in the memory interface block to DFP 320 or one of DSP's 300 buses.

Detailed Description Text (122):
DFP 320 can only access one section of the RAM memory 350 per request. In order to obtain a grant to a different section of the memory 350, DFP 320 must remove the bus request, wait for bus grant to be removed, change the memory address BMA[14:0], and then re-request the bus. The bus request signal (BR) is used to request the memory bus for the section of the memory referenced by the BMA address. DSP 300 will grant DFP 320 the memory block requested by asserting BGRANT. This grant will occur on instruction boundaries, the same as in the 2171. DSP 300 will continue to run if the GO mode is enabled until it needs to access the section of RAM granted to DFP 320. If the GO mode is not enabled, DSP 300 will stop as in the 2171. The GO bit should always be enabled on the platform. The GO MODE is MSTAT[6]. The bus request and grant function of the 2171 are quite different because the cycles to all sections of memory are the same and are not accessed with external memory cycles.

Detailed Description Text (123):
DSP's 300 bus grant hang signal, NBGH, can be used to remove DFP 320 from the memory section is currently has granted. The NBGH signal will be output from DSP 300 on

phase 2. This is the same phase as the NBG output. If NBG is used asynchronously to begin a RAM access cycle, this cycle should be finished before implementing the bus grant hang. If the NBG and NBGH are latched and used on CK12 trailing edge then the RAM access can be saved until after the hang wait. When the NBGH signal is active, DFP 320 will remove the NBR signal immediately or at the end of the next instruction cycle, finishing the one in progress as necessary. The NBR will be removed on phase 1. DFP 320 will then not request any block of the RAM again until 32 times the BGH wait plus one as defined in the control register, CTLREG[2:0]. One wait equals one instruction cycle. Therefore CTLREG[2:0] provides a 32 to 256 instruction cycle wait.

Detailed Description Text (124):
This RAM bus architecture uses some of the signals in DSP's 300 IOC block previously used for external/internal memory functions. The IOC block of DSP 300 has been changed to implement this RAM architecture. However, all DSP RAM memory is accessed without extended instruction cycles. This architecture allows:

Detailed Description Text (126):
2) fewer cycles/access for the host to RAM interface because 24 bit words are written in one cycle and the address does not have to be written first to an address pointer as in the 2181 IDMA

Detailed Description Text (127):
3) memory that is sometimes considered as external can be accessed by DSP 300 without extended W/R cycles for multiple accesses/instruction cycle.

Detailed Description Text (129):
Data flow processor (DFP)

Detailed Description Text (130):
This part will describe the functions of the data flow processor. DFP 320 has the following functions:

Detailed Description Text (131):
1) DFP 320 is a parallel interface to DSP 300 using direct memory access. DFP 320 serves as a hardware interface to DSP's 300 interrupt and executive operating software.

Detailed Description Text (133):
3) It serves as a bus interface by having 3 custom interfaces for DSP 300, MCU, and HW ACC blocks 330, 335, and 340. These allow the stream processor 500 to work independent of the bus structure.

Detailed Description Text (136):
FIG. 12 shows a block diagram of the major functional blocks. DFP 320 interfaces to three different bus configurations in the platform; the micro-controller 310, DSP memory 350, and hardware blocks. Each interface contains a functional block to provide buffering to allow interfacing to the stream processor 500 16 bit bus width.

Detailed Description Text (137):
MCU 310 is the master of MCU 310-DFP bus. DFP 320 uses MCU 310BUFREG, MBUFSTRREG, and MINASTRREG registers along with an interrupt pin to let MCU 310 write or read data. Since MCU 310 interrupt may take significantly longer than other data transfers, data stream packets are buffered in MCU 310 DATA STREAM BUFFER, address FFFE.

Detailed Description Text (138):
The stream processor 500 can use a buffer to keep data stream packets in its own data buffer when transferring between addresses. This buffer space helps manage multiple data streams by separating the data transfers across different buses.

Detailed Description Text (140):
The Stream processor 500 is the main block containing a state machine to automatically transfer data via a programmed data stream. DFP 320 contains registers

480 that are programmable by MCU 310 to configure DFP 320 for managing the data streams. The register descriptions are given below. DFP 320 can manage (for this implementation) 32 streams. For each stream there is a set of 4 registers 480 to control the stream. The stream processor 500 uses registers 480 to transfer data between to addresses in packets. The amount of data in a packet is programmed in the stream control register. Each stream has a programmed buffer size. The stream processor 500 will keep track of the amount of data transferred on a data stream until the buffer size is met. It then considers the data stream inactive and interrupts either MCU 310 or DSP for another buffer. The stream processor 500 will transfer a packet of data for each data stream when the source and destination are ready. The definition of when a source or destination address is ready to transfer data is different for each interface. The interface handshakes are cover below. The stream processor 500 is designed to only require minimum of control during data transfer of an active data stream. It automatically takes care of addressing, incrementing addresses for DSP memory and keeping the address constant for a hardware block.

Detailed Description Text (141):
The stream processor 500 also transfers data between the interfaces for MCU direct reads and writes. The bandwidths for direct I/O and different data streams will vary. In order to insure a worst case latency, the stream processor 500 checks the following top level functions in round robin to determine what data should be transferred.

Detailed Description Text (142):
Top Level Stream Processor Events

Detailed Description Text (146):
The DSP 300 has provided a stream buffer, get the buffer info from the memory mapped registers. The stream processor 500 has latch Flag Out falling edge.

Detailed Description Text (148):
Within the data stream processing top level task are several checks the stream processor 500 must perform to manage the data flow. Each time through the top level list of events, the stream processor 500 is going to attempt to perform one data stream packet transfer.

Detailed Description Text (150):
Check if the stream processor 500's data buffer is full. If so, attempt to complete a packet transfer out of the buffer.

Detailed Description Text (153):
The stream processor 500 maintains a status on each data stream programmed. The status can be ON, OFF, ACTIVE, or INACTIVE as defined before. FIG. 13 shows a flow of the status of each data stream.

Detailed Description Text (154):
DFP to DSP: Hardware Interface

Detailed Description Text (155):
The data flow processor (DFP) uses the RAM to move data to and from DSP 300. It interfaces to DSP 300 to request a memory block of the program or data RAM and to pass data and control information to and from DSP 300. DFP 320 and DSP share the following signals:

Detailed Description Text (156):
DSP Interface Signals

Detailed Description Text (160):
FI--The FLAG-IN pin is used by DFP 320 to inform DSP 300 that it is reading registers mapped to DSP memory. The FLAG-IN will be an output of the data flow processor to the FLAGIN signal on the schematic page `dspcore`, where it is an input to the sequencer (FIG. 14). (NOTE that the signal name is FLAGIN but it functions as FLAGIN.sub.-- as seen by DSP's 300 software.) The FLAGIN signal should not be connected to the DR1 pin, as on the AD12171. FLAGIN is mapped to DFP 320's CTRLREG

for read only.

Detailed Description Text (161):
FO--The FLAG-OUT pin is used by DSP 300 for three purposes. When FO is high it will keep DFP 320 from accessing the memory mapped registers. When FO transitions from 1=>0 it indicates to DFP 320 that the IRQ2 routine has ended. When not the IRQ2 routine, a negative edge on FO informs DFP 320 that a new DSP memory start address and DSP memory data block size are available. DSP 300 will set FO to 1 prior to reading or writing data from memory mapped registers. DSP 300 will set FO to 0 at the end of the IRQ2 routine or when done providing data through the memory mapped registers to DFP 320. FO will be latched by DFP 320 using DFP 320 clock CK12. DSP 300 must hold FO=1 for at least 2 instruction cycles. DFP 320 may access the memory mapped registers after FO=0. DSP's 300 signal `FLAGOUT` from schematic page SP1 should be sent to DFP 320 as an input and mapped to DFP 320's CTRLREG for read only.

Detailed Description Text (165):
RAM Interface Signals

Detailed Description Text (166):
NBR--The BUS REQUEST is used to request one section of memory, as determined by the BMA[15:0]. NBR is an active low input to DSP 300.

Detailed Description Text (167):
NBG--The BUS GRANT is used to control a section of memory, as determined by the BMA[15:0], for use by DFP 320. NBG is an active low output of DSP 300.

Detailed Description Text (168):
NBGH--The BUS GRANT HANG is used to allow DSP 300 to request the bus back from DFP 320 for the section of RAM granted to DFP 320. DFP 320 must remove NBR and wait a fixed number of cycles, as programmed in the control register in BGHW[2-0], before re-requesting the RAM bus for ANY section. NBGH is an active low output of DSP 300.

Detailed Description Text (169):
NB[23:0]--This is the bi-directional data bus for RAM interface. It is used to read and write to program and data memory. For bus interface access to data memory, NB[15:0] bits are used for data. When DFP 320 is reading the data memory, NB[23:16] are not driven by the RAM interface and are high impedance.

Detailed Description Text (170):
BMA[15 5:0]--This is the 16 bit address bus used to drive program and data memory selection and addressing. The address can only change when NBR or NBG are not active. This bus is an output of the bus interface. Bit 15 is used to access one of two optional 16K program memory pages. Bit 14 is used to access program or data memory. Bits 13:0 are used to access the 16K memory space where the 2 msbs are used to reference the appropriate memory block.

Detailed Description Text (171):
DFP.sub.-- NPC--bus interface RAM pre-charge signal. This must be driven by DFP 320 for RAM access. This signal is low during pre-charge.

Detailed Description Text (172):
DFP.sub.-- WR--bus interface RAM write. This signal is high for RAM writes and low for RAM reads. This is driven by DFP 320 for RAM accesses.

Detailed Description Text (173):
The timing diagrams in FIG. 15 and FIG. 16 show the interface for writing or reading for one and two data words respectively.

Detailed Description Text (179):
NBGH is not shown in the above timing diagrams. This is a DSP clock phase 2 signal that will occur on the same clock edge as a bus grant. See the above text description on NBGH.

Detailed Description Text (181):

DFP 320 will need to communicate with DSP 300 for transfer of all data. This includes commands for control and data for signal processing. DSP's 300 code to interface with DFP 320 will be implemented in the interrupt routine associated with IRQ2 and in DSP's 300 software that processes the information provided by the data flow processor. The reference to DSP 300 below is intended for both of these. Below is a description of the communication between DSP 300 and DFP 320

Detailed Description Text (183):
1) For DFP writes to DSP memory, except for the memory mapped registers, DFP 320 sets the BR signal of DSP 300 and expects BG on the next cycle. The handshake between DSP 300 interrupt routine and DFP are not involved.

Detailed Description Text (187):
DFP 320 will read or write the registers using the BUS REQUEST, BR, and directly drive the address and data bus to the memory.

Detailed Description Text (195):
1) DSP 300 will be able to initiate data flow through the signal interface to DFP 320. This acts as an interrupt to the stream processor 500. All data flow from algorithms or the OS routines are through host software reading or writing memory directly or through data streams. The flow of data through DFP 320 is mainly controlled through DFP 320 having access to buffers in DSP's 300 memory. DSP's 300 EXEC routines will provide these buffers. This outlines the handshake routine when DSP's 300 software wants to give information to DFP 320 for data streams, stream status, or interrupts.

Detailed Description Text (199):
DSP 300 will update one function at a time. If several functions need to occur such as the providing buffers for 2 data streams and updating the stream status, each will be provided separately. It is important to keep the interrupt routine and interface as efficient as possible. The data flow through the blocks works more efficiently and with less latency overall by letting data flow in the simplest manner. Having to read and process all registers slows the interface down for the most frequently used tasks.

Detailed Description Text (207):
When MCU 310 performs a write to DFP 320, DFP 320 will complete the write cycle and store the data. DFP 320 will then write the data to its own internal registers or gain access to DSP's 300 memory or a hardware block via DSP's 300 memory bus or the HW ACC bus resp. If a MCU write cycle is begun and DFP 320's internal stream buffer is full, is will not immediately acknowledge the write cycle. This condition should not occur if the buffer is large enough.

Detailed Description Text (209):
When MCU 310 performs a read cycle to DFP 320, DFP 320 will need to complete the process it may currently be performing. On average it will be available nearly immediately. DFP 320 will first decode the address, gain bus access if necessary, and then fetch the data. MCU 310 will need to wait for the data to have the read cycle completed. This will vary depending on the address location. DFP 320's address should be available for the 1.sup.st data cycle. There will be some delay for the HW ACC and DSP memory addresses.

Detailed Description Text (210):
MCU-DFP Interface for Data Streams

Detailed Description Text (211):
MCU 310 and DFP may transfer data through data streams programmed in DFP 320. Data streams may be flowing to or from MCU 310. All data will use a common buffer for MCU 310 interface, MCU 310 data stream buffer (MCUBUFREG). DFP 320's stream processor 500 will interrupt MCU 310 when a read or write cycle is needed to transfer data for one of the data streams. The handshake uses some of the registers listed in the part on DFP Register Descriptions. The interface for both situations, data to MCU 310 and data from MCU 310 are listed here.

Detailed Description Text (245):

DFP 320 performs reads and writes to all the hardware blocks over a common bus. There are two type of data transfers between MCU 310 and the hardware blocks. These are direct reads/writes by MCU 310 and data flow via a programmed data stream. The addresses in the hardware block used for data streams are generally the data input and output registers. Each data input or output register should be able to handle a packet size appropriate for the needed data bandwidth. This packet length, in number of 16 bit words, needs to be specified and is used by software. When a hardware block acknowledges a read or write cycle with the ARDY signal, it must drive or receive the entire packet without additional wait cycles. Each hardware block could have many input and output buffer lengths by specifying programmability in the control register.

Detailed Description Text (246):
When DFP 320 begins a write cycle to a HW block address, the HW block will decode the address and assert the ARDY signal. The HW block will then latch the data on the next cycle.

Detailed Description Text (247):
When DFP 320 begins a read cycle to an address, the HW block will assert ARDY to drive the data and complete the cycle.

Detailed Description Text (249):
DFP 320 acts as the bus master, however, each hardware block can request the flow of data to the block through request lines to DFP 320. Each hardware block has one request signal for each DFP address in the block that can be used as a source or destination of a data stream. The following list is the handshake sequence for DFP 320 to hardware blocks.

Detailed Description Text (252):
Any time the buffer at the data input address is completely empty or the buffer at the data output address is completely full, the request line for that address is asserted. This causes DFP 320 to service this stream. The request line should stay asserted until a read or write to the address is decoded and the ARDY is given. The request should go low before the end of the cycle.

Detailed Description Text (254):
For data out, DFP 320 will read the source address programmed for the stream number written to the request's HRQ<0-31>REG. DFP 320 will write the data to an internal buffer in the stream processor 500.

Detailed Description Text (255):
For data in, DFP 320 will get the data and write it to the stream processor 500 buffer and then write the data to the destination address programmed for the stream number written in the request's HRQ<0-31>REG.

Detailed Description Text (259):
This part describes data streams are processed. While the above parts covered just the hardware interface, this part lists details of all the blocks for some data stream examples.

Detailed Description Text (263):
DFP 320 detects an inactive stream (see Part (TBD) DATA BUFFERS for how this is done). It will complete a RAM access if one is in progress until the RAM bus is not requested.

Detailed Description Text (265):
DFP 320 will set the IRQ2 pin of DSP 300 high for 1 instruction cycle. The IRQ function should be programmed to be edge sensitive in DSP 300. Programming and edge sensitive IRQ also allows use of software interrupts via DSP 300 IFC register. DFP 320 will wait until DSP 300 IRQ2 routine is complete by waiting until it detects a 1=>0 transition on FO. During this time it will not request RAM access.

Detailed Description Text (300):
When the HRQ<>REG for the HW ACC's output data address is high, DFP 320's stream processor 500 will read the address for the number of words in a packet programmed

in the data stream's control register.

Detailed Description Text (301):
DFP 320 will write the data into the stream processor 500 data buffer.

Detailed Description Text (322):
When the HRQ<>REG for the HW ACC's input data address is high, DFP 320's stream processor 500 will wait until MCU 310 DATA STREAM BUFFER is empty (if necessary) and write the stream number to the MINASTRREG. Note that the average data rate is controlled by the hardware accelerator.

Detailed Description Text (339):
DFP 320 will get the stream number from the MINASTRREG and write the data to the destination address. If the stream number is 0, DFP 320 will continue processing data streams. If the stream number is different from the requested stream, DFP 320's stream processor 500 will put the stream in queue to have the data written to the proper address. The data will sit in the stream processor 500 internal buffer until the HRQ<>REG for the stream is active.

Detailed Description Text (340):
DFP 320's stream processor 500 will give priority MCU 310 W/R accesses to addresses in DFP 320's address space except for MCU 310 INTERFACE BUFFER address(es). DFP 320 will give an equal priority to all other data flow.

Detailed Description Text (366):
Hardware accelerator (HW ACC) functions are hardware blocks designed to do a specific operation or several operations, based on accessible configuration registers. Functions typically accelerated are those that are implemented more efficiently in specific hardware than as DSP firmware. All accelerated functions interface with DFP 320 to send and receive data on a common bus. DFP 320 is the bus master for the HW ACC bus. The HW ACCs can be accessed using DFP 320 by reads and writes or data streams. The control of the HW ACC block is done through data reads and writes to registers in each hardware accelerator. The signal interface between the HW ACC block and DFP 320 is used for bus control only. Each HW ACC will begin an operation when the START bit in the control register is written high. Starting of the accelerator can come from any of DFP 320 input sources and is directed to the particular accelerator using the address of its control register. The input data and output data of the accelerated function will typically use the data memory associated with DSP 300 as buffers but could use other valid addresses within DFP 320 address space. Although the HW ACC block is a slave on the bus, it is responsible for controlling the data rate for input and output data in both directions through two signals to DFP 320. DFP 320 function provides data flow to and from the HW ACC through a data stream at the pace of the accelerator's processing. The HW ACC's clock may be run at a different rate than DFP 320.

Detailed Description Text (369):
Once DFP 320 has data streams programmed for the input data and output data registers, the HW ACC control register can be started by writing a 1 to the START bit. The HW ACC will begin processing a task. Each HW ACC will have input and output registers to buffer the data for reads and writes. The size of these HW ACC bus buffers will depend on the data needs of each accelerator. The size of the buffer will be used by the software on MCU 310 to program the data stream buffer size. When DFP 320 receives ARDY, it will expect to write or read to each HW ACC as many words as programmed in the data stream buffer size for the data stream. The size can be adjusted for each HW ACC to control the data throughput and overhead of the data transfers.

Detailed Description Text (370):
When the HW ACC input buffer is empty, the request signal for the input data buffer, REQ.sub.-- HW#, is asserted. DFP 320's stream processor 500 will eventually perform a write to the HW ACC#'s input data address. When the HW ACC# is ready for the write cycle, it pulls down the ARDY signal. DFP 320 will write the number of bytes programmed in the data stream in one write cycle.

Detailed Description Text (371):

When the HW ACC output buffer is full, the request signal for the output data buffer, REQ HW##, is asserted. DFP 320 will perform a read of the HW ACC#'s output data address. When the HW ACC# is ready to complete the entire read cycle, it pulls down the ARDY signal. DFP 320 will read the number of bytes programmed in the data stream in one read cycle.

Detailed Description Text (376):
Hardware Accelerator Functions--Interface Signals to DFP

Detailed Description Text (377):
ADDR.sub.-- DATA[7:0]--address and data bus, bidirectional. All addresses are two bytes long, sent lower byte first. The data is formatted by DFP 320 to 16 bit words for transfer to any other block.

Detailed Description Text (378):
ARD--Accelerator bus ReaD, DFP output. DFP 320 will cause the ARD to go inactive during the last data byte. The number of words transferred

Detailed Description Text (380):
AWR--Accelerator bus WRite, DFP output. DFP 320 drives the AWR signal inactive during the last data byte.

Detailed Description Text (381):
ARDY.sub.-- L--Accelerator bus ReaDY, common HW ACC output. Each HW ACC will pull down the ARDY signal when the address has been decoded and the particular HW ACC block is ready to complete the read or write cycle. The ARDY should go inactive during the 1.sup.st data byte.

Detailed Description Text (383):
The HW Accelerator--DFP interface is shown in FIG. 18.

Detailed Description Text (397):
The timer block will interface to DFP 320 via the HW ACCEL bus. The bus transfers 16 bit data on an 8 bit bus.

Detailed Description Text (399):
The timer will have to compare the counts for each of the tasks programmed against the TDMA counter. As each event count is reached, the timer will perform two types of event triggers. One is to send data over the hardware accelerator bus. The other is to provide a direct signal from block to block. To send data, the timer will retrieve the data sent for the event during programming and send the destination address and data using a DFP read cycle of the data output register, DATAOREG. The timer will drive the destination address and data programmed for the event onto the hardware accelerator bus. For events that have a direct output, such as the FRAME signal, a 1 bit wide pulse will be output on the signal. Each of the direct outputs have their own address and register to program the TDMA count.

Detailed Description Text (410):
Tasks programmed off a count in the TDMA timer are written to this register. The following data values are written in order to this register is a single write cycle. Each data value is sent in 2 bytes.

Detailed Description Text (416):
This register is read by DFP 320 when the REQ pin is active. The timer will provide the 2.sup.nd and 3.sup.rd words of data written when the event was programmed during the read cycle. Each data value will be provided as a 16 bit word according to the HW ACCEL bus of DFP 320.

Detailed Description Text (418):
DSP 300 contains two main routines for controlling the operation of algorithms and interface. The interrupt routine interfaces to DFP 320 and performs handshaking to transfer control of W/R to DFP 320 control registers memory mapped in DSP data memory. The interrupt routine is designed to execute quickly transferring information to DFP 320. The interrupt routine should be about 100 instructions and run <100 cycles average per interrupt.

Detailed Description Text (422):
The executive (EXEC) routine has several sub-routines that perform memory management of data buffers used in the task schedule, execution of the task schedule from a task list, and execution of commands. The interaction of the interrupt and EXEC sub-routines in the OS are shown in FIG. 21. The flow charts for the individual EXEC subroutines are shown FIG. 22-FIG. 25.

Detailed Description Text (425):
The memory manager keeps track of all the buffers for data streams. Buffers in DSP data memory for algorithms will have predetermined addresses. The memory manager interfaces with DFP 320 and the task scheduler routine, through the buffer queue, to transfer data memory buffers. The memory manager will handle up to two buffers per data stream for double buffering.

Detailed Description Text (427):
The flow chart in FIG. 22 refers to the data stream buffers in DSP memory as available, busy, or used. Each buffer is passed between DFP 320, memory management routine, and task scheduler for transferring data. The meaning of each of the three terms depends on if a stream is sending data into or out of DSP 300.

Detailed Description Text (428):
Streams flowing out of DSP 300 have buffers made available to the memory manager by the task scheduler when the buffer is full of data. Buffers are buy after the buffer address and size has been given to DFP 320 for data stream packet transfers. The buffer stays busy until DFP 320 requests another buffer for that stream number, which means the previous one is completely read. At this point the buffer is used. The memory manager will give the buffer to the task scheduler.

Detailed Description Text (429):
Streams flowing into DSP 300 have buffers made available to the memory manager by the task scheduler when the buffer is empty. Buffers are busy after the buffer address and size has been given to DFP 320 for data stream packet transfers. The buffer stays by until DFP 320 requests another buffer for that stream number, which means the previous one is completely written. At this point the buffer is used. The memory manager will give the buffer to the task scheduler.

Detailed Description Text (433):
The TASK LIST is a buffer used by the TASK SCHEDULER to maintain the status of all the programmed tasks and scheduling parameters. The TASK LIST is built and maintained by the TASK.sub.-- PROG routine. The task list is parsed often when the EXEC runs and contains direct pointers for efficiency. The TASK.sub.-- PROG routine is used to interpret task numbers and order as programmed by MCU 310 into a table for the task scheduler to access.

Detailed Description Text (434):
The task list contains functional information on task order and related data stream numbers and buffer addresses. The TASK LIST is referenced whenever the TASK SCHEDULER runs. For each task programmed there can be many streams associated with the task. These would be used for signal processing data buffers as well as control oriented data passed between functions. For example an algorithm could be initiated by the TASK SCHEDULER when the input data is available and the timer count reaches a programmed value. One buffer would be used to write the input data and a second 1 word buffer for receipt of a word written by DFP 320 from the timer. Each data stream can have one or two buffers associated with it. If only one is used the second is written to 0. A control word is associated with each buffer in the task list. Two task numbers are needed in the control word to determine when the buffer is made available to the memory manager and a task number to determine when to leave the buffer status as used. This does two things, it allows DFP 320 and memory manager to work more efficiently by only processing buffers and data for streams that are being used. It also allows DSP data memory reuse.

Detailed Description Text (435):
At the end of the entry for each task data will be written to list the items that gate the starting of the task. These would typically include stream buffer status

and the status of another task(s) (completed, in progress).

Detailed Description Text (442):
a data buffer status changing from `busy` to `used`. This means an input buffer has become full, or an output buffer has been read.

Detailed Description Text (443):
the TASK PROGRAM sub-routine has run and modified the TASK LIST

Detailed Description Text (444):
DSP Exec--Command Interface Subroutine

Detailed Description Text (445):
The command interface routine is used to interpret data passed between MCU 310 and DSP 300. The list of functions would depend on the system implementation. Basically any information outside the task scheduling and data flow needs to be handled. These commands could be used to by MCU 310 to control specific algorithms, interrogate status, or program DSP registers.

Detailed Description Paragraph Table (1):
RESET.sub.-- resets the entire device in addition to the 2171 reset functions MMAP This pin is removed and the MMAP function is fixed at MMAP=1. Upon reset, the boot sequence is initiated by software. There is no difference in booting from a reset (hardware or software) or from software after a reset has occurred. This allows the software to have full control. BMODE The boot mode is fixed to boot through the data interface, BMODE=0. IRQ1.sub.-- This pin exists on the device. It is also mapped to the interrupt register in DFP 320 regardless of the state of the system control register Ox3FFF. The reset condition for both SPORTs is disabled and the interrupt and flag pins to be active on the SPORT1 pins. The function is the same as the 2171. When the system control register Ox3FFF enables SPORT1 and the SPORT1 CONFIGURE bit is set to 1 the TFS1 function is the same as the 2171. IRQ0.sub.-- This pin exists on the device. It is also mapped to the interrupt register in DFP 320 regardless of the state of the system control register Ox3FFF. The reset condition for both SPORTs is disabled and the interrupt and flag pins to be active on the SPORT1 pins. The function is the same as the 2171. When the system control register Ox3FFF enables SPORT1 and the SPORT CONFIGURE bit is set to 1 the RFS1 function is the same as the 2171. FO The flag out pin functions as in the 2171 and is controlled by the system control register Ox3FFF depending on the state of the SPORT1 ENABLE and SPORTl CONFIGURE bits. It is also mapped to the IRQ register in DFP 320 and is an input to DFP 320. FI The flag in pin functions as in the 2171 with respect to the 2171 software. However it is a direct hardware input from DFP 320 block. Therefore, the state of FI is available to software regardless of the state of the SPORTl CONFIGURE bits. It is also mapped to the IRQ register in DFP 320 as a read only bit. FL2-0 The flag bits are connected to DFP 320. Their functions are the same as the 2171. There states are mapped to the IRQ register in DFP 320. PWD.sub.-- The power down function can be initiated by the pin or through software by setting the POWER DOWN FORCE control bit in the ANALOG AUTOBUFFER/POWERDOWN control register. A hardware power down causes the entire device to enter a power down state. The 2171 power down function is the same. The remainder of the device must enter a low power state where clocks are stopped at the appropriate phase. DFP 320 control remains active to receive input. The power down mode can be exited through the software initiated power down. Since all bus activity is buffered to the core, DFP 320 will generate a power down of the non- core circuitry after detecting the write of the POWER DOWN FORCE control bit to the ANALOG AUTOBUFFER/POWERDOWN control register. PWDACK The power down acknowledge pin incorporates the valid powerup condition of the core as well as the entire device. The logic states are the sarne as the 2171. The clock generator's outputs are valid when the PWDACK pin is driven low. BR.sub.-- The bus request is used for interface between DSP 300 and DFP 320 to allow DFP 320 to access the memory directly. It is a DFP output. DFP 320 will assert BR.sub.-- to perform reads and writes of program and data memory. BR.sub.-- is used during the boot sequence. The 2171 `GO` mode works as normal. BG.sub.-- Bus grant is output from DSP 300 to DFP 320. Its function is similar to the 2171 except the bus is granted to DFP 320 for the section of RAM requested and the RAM is on chip. BGH.sub.-- The bus grant hang pin is output by DSP 300 to DFP 320. Its function is similar to the 2171. It is used to allow DSP 300 to

request the memory bus when it is currently granted to DFP 320. DSP 300 can assert the BGH.sub.-- signal to DFP 320. In the 2171, it is designed for a multiprocessor environment. Here it can be used to allow DFP 320 to relinquish the bus to DSP 300 and request it back after a programmable delay.

Detailed Description Paragraph Table (4):

DSP DSP DSP 2] 1] 0] IRQ2 IRQ1 IRQ0 This register is used for bus interface and 301 clocking control. DSP FI - This bit is a read only bit. The bus interface output, FLAG IN, is mapped to this bit. Writing to this bit has no effect, DSP FO - This bit is a read only bit. DSP 300 output, This bit controls a mux for the IRQ2 input of DSP 300. When this bit is 0, the IRQ2 signal operates per the normal 2171 functions. When this bit is 1, the IRQ2 signal is controlled by DSP 300 IRQ2 bit in the 301 IRQREG. However, DFP 320 can not function normally because it can not cause an interrupt to DSP 300. IRQ1 MUX - This bit controls a mux for the IRQ1 input of DSP 300. When this bit is 0, the RQ1 signal operates per the normal 2171 functions. When this bit is 1, the IRQ1 signal is controlled by DSP 300 IRQ1 bit. However, the configuration state of the SPORT1 must be set to 0 for interrupts and flag mode, System Control Reg (ox3fff)[10]=0, for the IRQREG to act as an input. IRQ0 MUX - This bit controls a mux for the IRQ0 input of DSP 300. When this bit is 0, the IRQ0 signal operates per the normal 2171 functions. When this bit is 1, the IRQ0 signal is controlled by DSP 300 IRQ0 bit. However, the configuration state of the SPORT1 must be set to 0 for interrupts and flag mode, System Control Reg (ox3fff)[10]=0, for the IRQREG to act as an input. BGH wait time, in multiples of 32 instruction cycles, for DFP 320 to hold off requesting any block of the RAM after NBGH is asserted. Each bit of the wait time has weight of 32 instruction cycles giving the delay a range from 32-256 instruction cycles. DSP IRQ2 - Writing a 1 to this bit will interrupt DSP 300 for IRQ2. Writing a 0 cause the IRQ2 input to go low. DSP 300 must be pro- grammed for edge sensitive interrupts per the bus interface operation. Therefore, this interrupt will be edge sensitive. This interrupt is normally used by DFP 320 for DSP 300 interface. When used DFP 320 should not be processing streams. DSP IRQ1 - This bit is mixed to the input of DSP 300 IRQ1 function when selected by CTLREG[7]. Writing a 1 to this bit will interrupt DSP 300 for IRQ1. Writing a 0 cause the IRQ1 input to go low. DSP 300 must be programmed for edge sensitive interrupts per the bus interface operation. Therefore, this interrupt will be edge sensitive. This interrupt is a function of the System Control register, the SPORT1 must be con- figured for interrupts and flags. DSP IRQ0 - This bit is muxed to the input of DSP 300 IRQ0 function when selected by CTLREG[7]. Writing a 1 to this bit will interrupt DSP 300 for IRQ0. Writing a 0 cause the IRQ0 input to go low. DSP 300 must be programmed for edge sensitive interrupts per the bus interface operation. Thefefore, this interrupt will be edge sensitive. This interrupt is a function of the System Control register, the SPORT1 must be configured for interrupts and flags.

Detailed Description Paragraph Table (5):

15 14 13 12 11 10 9 8
RESET.sub.-- SHUTD CLKO NBR L N K REL
7 6 5 4 3 2 1 0
DSP PIN DFP RAM RAM TEST TEST MUX SCAN BIST
SCAN CLK PIN CON- AC- TROL CESS RESET.sub.-- L - This bit is the software reset. Writing a 0 to this bit will perform a reset of DSP 300 & base band hardware except for MCU 310. For DSP 300, this bit is DSP 300 RESET.sub.-- L bit located in the ADI 2171 HIP register space relocated to DFP 320 block. A 1=>0 transition will be latched by the main clock in signal directly from the pin and used to generate a reset pulse for the hardware. DSP 300 clock block is also reset using this method. The register is set back to a 1 after the reset is completed. SHUTDN - When this bit is 1, it causes all base band HW except MCU 310 to enter the shut down low power state. CLKOK - This bit is a read only bit. The clock generator signal, CLKOK, is mapped to this bit. It is used to start the clock generator. When CLKOK is low, the clock generator is not stable. When it is 1, the clock generator is stable and all clocks are present, depending on the other inputs to the clock generator block. NBR RBL - Bus request release. This bit reset to 0 and causes the data flow processor to assert bus request and hold it until a 1 is written to this bit. The NBR signal to DSP 300 is an OR ofthis bit and the normal NBR output from the bus interface control. DSP TEST PIN - enables DSP 300 test mode

PIN MUX CONTROL - controls pins muxed for observability of the embedded DSP DFP SCAN - Scan enable for DFP 320 RAM BIST - self test for DSP 300 RAMs RAM SCAN - boundary scan for RAM TEST CLK ACCESS - enable for direct control of clocks

Detailed Description Paragraph Table (6):

##STR1## DFP 320 uses a buffer to MCU 310 for all data streams. This buffer allows DFP 320 - MCU data transfer to handle clock differences and keeps DFP 320 processing from slowing down MCU W/R accesses. The depth of the buffer will depend on the particular design. The depth of the buffer will effect the overhead and band- width of MCU 310/DFP interface Data for data streams to/from MCU 310. The stream number is given in the INACTIVE STREAM NUMBER register and read by MCU 310.

Detailed Description Paragraph Table (8):

15 14:8
SS ENA SS
COUNT 7 6
5 4 3 2 1 0
BP1L[2]
BP1L[1] BP1L[0] BP2L[2] BP2L[1 BP2L[0] GO FREEZE]
SS ENA - Single Step ENAble. When 1 the bus interface will allow the clock generator to run for SS COUNT cycles and stop it with the FREEZE bit. The clock should be frozen before this is enabled. When the SS COUNT is reached, this bit is cleared. SS COUNT - Single Step COUNT. When SS ENA is written the bus interface will remove FREEZE for this number of instruction cycles. If a break point occurs first, the FREEZE will be asserted on the break point. Break Point 1 Loop. Each time a break point field is enabled, the bus interface will loop on the break point condition before asserting FREEZB and removing the BP(1 or 2) ENA. Break Point 2 Loop. Each time a break point field is enabled, the bus interface will loop on the break point condition before asserting FREEZE and removing the BP(1 or 2) ENA. GO - Writing a 1 to this bit causes the bus interface to remove FREEZE if it was asserted from a debug mode. If FREEZE is asserted because of SHUT DOWN or POWER DOWN, FREEZE will not be removed. FREEZE - This is a read only. The bus interface FREEZE output to the clock generator is mapped to this bit

Detailed Description Paragraph Table (9):

##STR3## BG - bus grant. If high, the bus grant should be active which means the bus interface is currently accessing the address. If bus grant is low, DSP 300 is accessing the current RAM address. W - High for a break on a write. R - High for a break on a read. FL gate - High when the break should occur only on Flag Out. Otherwise the first event, RAM access or Flag Out will couse the break point. FL1 - High for a break on Flag Out. Flag Out is asserted when DSP 300 exits the IRQ2 routine. FL gate - High when the break should occur only on Flag Out 1 or 0. Otherwise the first event, RAM access or Flag Out will cause the break point. FL1 - High for a break on Flag Out 1. FL0 - High for a break on Flag Out 0.

Detailed Description Paragraph Table (10):

15 14:8
BP ENA ADDR[14:8]
7:0
ADDR[7:0] BP ENA - Enables the break point. The bus interface will clear this bit when the break point is met after the break point loop is counted down. DSP memory address, program and data. BPF2[7:0]NT FIELD 2 CONTROL Identical to break point field 1. BPFA2[15:0] FIELD 2 ADDRESS Identical to break point field 1. BPF3[7:0]NT FIELD 3 CONTROL Identical to break point field 1 except there is no break point looping. BPFA3[15:0] FIELD 3 ADDRESS Identical to break point field 1 except there is no break point looping. SSTATREG [15:0](Default:TUS `h0000) memory mapped This register is for reporting overflow and underflow errors in the data stream flow. Any bit set high in this register causes the IRQ[6] to be set. Each STREAM # for streams 15-0 are mapped to the bits of the registers. The bits are set by DFP 320 and cleared by host software writes. When DFP

320 is processing an active stream and data overflows or underflows, it will set the bit corresponding to the STREAM #. Since streams are unidirectional only one condition, underflow or overflow, is likely for each data stream.

Detailed Description Paragraph Table (11):

##STR4## This set of registers is used to program each data stream service request line from a hardware block to one of the stream numbers. For the specifi- cation there are 32 request lines. The actual number would depend on the design. This register is written once when each stream is programmed that involves a data stream to a hardware block Stream number. One of the stream numbers known to the stream processor 500. For this specification, 0-31. The stream number written to a particular hardware accelerator block's re- quest line would be the stream number programmed to use that address as a source or destination for data flow.

Detailed Description Paragraph Table (16):

##STR8## ##STR9## PACKET SIZE[4:0]]- Number of 16 bit words transferred by DFP 320 in the data stream at a time. For data to and from MCU 310, this number should not exceed MCU 310 BUFFER size (32). For hard- ware blocks, this number should equal the size of the data buffered for each read or write access. PACK - for use with the program memory only. When Pack=1, the 24 bit data is packet to be transferred in 4 bytes. When Pack = 0, the program memory data is either zero padded or sign extended depending on bit [6]. SX - sign extension. When this bit is 1, a 16 bit word is sign ex- tended to 32 bits. When this bit is 0, the valid work should be zero padded on the msbs to form a 32 bit word. DIR - The data stream direction when the stream involves DSP 300. When this bit is 1, DSP 300 is the source of the stream. A 0 when DSP 300 is the destination. HOST/DSP BUF INT - host/DSP memory buffer interrupt. When this bit is 1, DFP 320 will cause an interrupt to MCU 310 when the current memory address generated by the buffer interface has met DSP 300 memory start address + DSP memory buffer size. When the bit is 0, the buffer interface will interrupt DSP 300 for a new start address and memory buffer size. HOST/DSP START - When this bit is 1, DSP 300 Memory Start Address and DSP Memory Buffer Size fields are valid. When the bit is 0, DFP 320 should proceed to get DSP 300 memory start address from DSP 300 when the STREAM ON bit transitions from 0=>. Until DFP 320 gets a valid DSP Memory Start Address and DSP Memory Buffer Size, the data stream is considered inactive. STREAM ON - This bit must be set high to cause the stream pro- cessor 500 to control whether the state of the data stream is in-active or active. This bit should be cleared to turn the stream off. However, a stream that has become inactive will not cause and processing cycles and does not have to be turned off.

Detailed Description Paragraph Table (17):

##STR10## Stream's source start address. This value is used by DFP 320 as a starting address for writing or read- ing data to the source of the data stream, depending on the address. The stream processor 500 increments the address for each transfer of a word to or from DSP 300 memory. For addresses to the hardware accelerator's space, DFP 320 will not increment the address while transferring data. The last data transfers for a data stream occurs when the address is equal to DSP 300 memory start address + DSP memory buffer size. When this happens, DFP 320 causes an interrupt to the host or DSP 300, depending on the SCTL[2] bit in the STREAM CONTROL register, SCTL.

Detailed Description Paragraph Table (18):

##STR11## memory start address. This] value is used by DFP 320 as a starting address for writing or reading data to the destination of the data stream. The stream processor 500 increments the address for each transfer of a word to or from DSP memory. For addresses to the hardware accelerator's space, DFP 320 will not increment the address while transferring data. The last data transfers for the data stream occurs when the address is equal to DSP 300 memory start address + DSP memory buffer size. When this happens, DFP 320 causes an interrupt to the host or DSP 300, depending on the SCTL[2] bit in the STREAM CONTROL register, SCTL.

Detailed Description Paragraph Table (21):

##STR15## ##STR16## DATAIN - Input data for processing. Data is written to this address by DFP 320. This address needs to be specified for the buffer size, in words, in the device specification. This number is used to program DFP 320 for the data stream buffer size. Once the ARDY signal is active following a write to this address, DFP 320 will write the complete buffer size programmed in one write cycle.

Detailed Description Paragraph Table (22):

##STR17## ##STR18## DATAOUT - Input data for processing. Data is read form this address by DFP 320. This address needs to be specified for the buffer size, in words, in the device specification. This number is used to program DFP 320 for the data stream buffer size. Once the ARDY signal is active following a read to this address, DFP 320 will read the complete buffer size programmed in one read cycle.

Detailed Description Paragraph Table (23):

Block Signals:

Data (7:0) I/O 8 bits for address and data ARD IN accelerator bus read AWR IN accelerator bus write ARDY.sub.-- L OUT accelerator bus ready SFT has decoded the address and is ready to complete the cycle. FRAME OUT 1 bit wide pulse, positive edge occurs at count loaded in FRMPOSREG REQ.sub.-- TMR OUT request for DFP 320 to read the data out register, DATAOREG. TIMER.sub.-- OUT1 OUT 1 bit wide pulse, positive edge occurs at the count loaded on a programmed event. TIMER.sub.-- OUT2 OUT 1 bit wide pulse, positive edge occurs at the count loaded on a programmed event. The MMUSFT interface is shown in Figure 19.

CLAIMS:

1. A programmable data flow processor operable for performing data transfers between a plurality of devices, the data flow processor comprising:

a plurality of ports each for coupling to a device;

programmable configuration registers which are operable to receive data for

programming the data flow processor for data transfers between selected ones of said plurality of ports;

a stream processor coupled to each of said plurality of ports and coupled to said programmable configuration registers, wherein the stream processor is operable to access data comprised in said programmable configuration registers and in response configure a plurality of data streams between

said plurality of ports, wherein each of said plurality of data streams is transferred between at least one source port and at least one destination port, and wherein said stream processor is operable to transfer said plurality of data streams between said plurality of ports;

wherein the data flow processor includes at least one buffer for temporarily storing data transferred between two devices; and

wherein the stream processor determines an ordering of transfers between said plurality of ports based on buffer availability.

2. The data flow processor of claim 1,

wherein each of the ports include information indicating availability for data transfer;

wherein the stream processor determines an ordering of transfers between said plurality of ports based on buffer availability and said port availability information.

3. The data flow processor of claim 2,

wherein the stream processor is operable to determine when data transfers for data streams are to occur.

4. The data flow processor of claim 3, wherein the stream processor is operable to determine when data transfers for data streams are to occur based on one or more criteria from the group comprising: stream processor buffer availability, port availability information, configuration register data; the amount of data being transferred for a respective stream, and ordering of the transfers.

5. The data flow processor of claim 2, wherein said programmable configuration registers comprise information indicating the priority of each of said data streams.

6. The data flow processor of claim 2, wherein, for each respective port, said port availability information comprises information indicating an availability of a buffer comprised on said respective port for transferring/receiving data.

7. The data flow processor of claim 1, wherein said stream processor stores information on an amount of data transferred for each of said data streams;

wherein the stream processor transfers a programmable amount of data for each of said plurality of data streams.

8. The data flow processor of claim 1, further comprising an address conversion unit coupled to said stream processor, wherein said address conversion maps addresses between said data flow processor and said ports.

9. The data flow processor of claim 1, further comprising a controller bus interface coupled to one of said plurality of data ports, wherein said controller bus interface couples to a controller, wherein said controller programs said configuration registers for performing said data stream transfers.

10. The data flow processor of claim 1, wherein one or more of said ports comprise serial ports.

11. The data flow processor of claim 1, wherein one or more of said ports comprise parallel ports.

12. The data flow processor of claim 1, wherein one of said ports is adapted for coupling to a digital signal processor.

13. The data flow processor of claim 1, further comprising a hardware accelerator bus interface coupled to one of said plurality of data ports, wherein said hardware accelerator bus interface is adapted for coupling to at least one hardware accelerator unit.

14. The data flow processor of claim 1, further comprising a memory bus interface coupled to a first one of said ports, wherein said first port is adapted for coupling to a memory.

15. The data flow processor of claim 1, wherein said programmable configuration registers comprise information regarding which device is interrupted when one of said data streams is complete.

16. The data flow processor of claim 1, wherein said stream processor transfers data by incrementing a source and a destination address for each of said data streams.

17. The data flow processor of claim 1, wherein said stream processor transfers data in packets.

18. The data flow processor of claim 1, wherein said data flow processor is operable to receive source and destination addresses for said data streams and an amount of

data to be transferred.

19. A programmable data flow processor operable for performing data transfers, the data flow processor comprising:

a plurality of ports each for coupling to a device;

programmable configuration registers which are operable to receive data for programming the data flow processor for data transfers between selected ones of said plurality of ports; and

a stream processor coupled to each of said plurality of ports and coupled to said programmable configuration registers, wherein the stream processor is operable to access data comprised in said programmable configuration registers and in response configure a plurality of data streams between said plurality of ports, wherein each of said plurality of data streams is transferred between at least one source port and at least one destination port, and wherein said stream processor is operable to transfer said plurality of data streams between said plurality of ports;

wherein the data flow processor includes at least one buffer for temporarily storing data transferred between two devices;

wherein each of the ports include information indicating availability for data transfer;

wherein the stream processor transfers a programmable amount of data for each of said plurality of data streams, wherein said amount of data is transferred in packets, and wherein an amount of data transferred in each packet is programmable;

wherein the stream processor determines an ordering of transfers between said plurality of ports based on buffer availability and said port availability information; and

wherein the stream processor is operable to determine when data transfers for data streams are to occur.

WEST

Free block

End of Result Set

Generate Collection Print

L50: Entry 2 of 2 File: USPT Aug 11, 1992

DOCUMENT-IDENTIFIER: US 5138696 A
**** See image for Certificate of Correction ****
TITLE: Printer provided with font memory card

Brief Summary Text (4):
The electrophotographic printer employs function formulae such as Bezier Mathematical Curves, Spline Mathematical Curves and the like which represent the outline of characters instead of the conventional dot matrix fonts (bit map fonts), and an outline font for storing therein connection points of the function formulas, and parameters and the like for controlling the connection points. Using this outline font, it is possible to change the sizes and formats of characters such as bold face and hollow outline with ease. However, the use of this outline font has the drawback of long processing times needed to generate bit map data based on this outline font. To solve the problem, a font cache memory is provided for avoiding the generation of the bit map font from the outline font when the same characters are reused, thereby to speed up the processing speed. That is, the first time that the bit map font is generated, the thus generated bit map font is stored in the font cache memory. When the same characters are printed again, they can be printed based on the bit map dated font in the font cache memory, thereby facilitating high speed printing.

Brief Summary Text (5):
Conversion of the outline fonts into the bit map fonts having the specific sizes is done using routines (and/or data) which are previously stored in the program for controlling the printer. Once the printer is turned on, the outline fonts are converted into the bit map fonts having the specific sizes which are instructed by the control program and stored in a font cache memory. When the printer (thereafter) receives a printing character code from a host unit such as a personal computer or word processor, the outline fonts will already have been converted into the corresponding bit map fonts.

Detailed Description Text (7):
The processor 4 generates bit map font data for printing based on the function expressions and the parameters when it receives printing character codes from the host unit 20. The font cache memory 6b is a cache memory for temporarily storing the thus generated bit map font data and forms a part of the working memory 6. The image memory 8 comprises a RAM (Random Access Memory) for storing printing data, e.g. for one page which are edited and imaged by the processor 4. The print engine 10, which carries out printing on printing sheets based on the printing data stored in the image memory 8, comprises a printing sheet feeding system, printing process means for electrophotographic printing and the like. The print engine interface 9 is an interface circuit which reads the printing data from the image memory 8 in accordance with the instruction from the processor 4 and transfers the thus read data to the print engine 10. The print engine interface 9 receives print control signals from the print engine 10, sends print control signals to the processor 4, receives instructions from the processor 4, and transfers instructions to the print engine 10.

Detailed Description Text (8):
The nonvolatile memory 11 comprises the memory and the like backupped by the battery as mentioned above and stores data for initializing the printer 1 when the power is

turned on. The font memory card 13 is composed of a nonvolatile memory and selects the bit font data stored in the font cache memory 6b or in the font memory card 13 itself based on the frequency of use thereof, and stores the bit map font data therein, described later.

Detailed Description Text (14):
The S-RAM 130 comprises several elements. Beginning at a starting address shown at the top of FIG. 3, S-RAM 130 includes a memory card identification (ID) number storage area 131 for storing user's information of the font memory card 13 and the like. A memory card type information storage area 132 is provided for storing data for distinguishing one type of memory card from other memory cards formed of a mask ROM or the like. Information about the storage capacity of the font memory card 13 may be contained in storage area 132. Next, a total font number storage area 133 is provided for storing a number called the total font number. This represents the number of types of fonts stored in the font memory card 13. The S-RAM 130 further comprises the following elements next to the total font number storage area 133: a font attribute address table area 200 for storing a managing number given to each font type stored in the font memory 13, the number of frequency of use, starting and ending addresses of the character attribute address table, described later, and a free area 134 as a spare area of the font attribute table 200. Following by the free area 134 are a character attribute address table 300 for storing a character code managing number corresponding to a character set table, a block address number representing the block where the bit map font data corresponding to the character code managing number is stored and the number of blocks to be used, a free area 135 as a spare area of the character address table area 300, a bit map font area 136 for storing various bit map fonts and a free block retrieve table area 400 for dividing the areas extending from the font attribute address table 200 to the bit map font area 136 every r bytes, r being an integer. Free block retrieve table area 400 also stores free block retrieve flags which represent information about the unused blocks among the bit mag font blocks.

Detailed Description Text (20):
As shown in FIG. 3, the character attribute address table 300 is the table representing the storage addresses of the bit map fonts corresponding to the font character codes specified by the combination of the font types (TF.sub.n) and sizes of characters (SZ.sub.n). Accordingly, the character attribute address table exemplifies, as illustrated in FIG. 5(A), the combination of the font types TF.sub.1 and the character sizes SZ.sub.1 in the table of FIG. 4. The character attribute address table 300 comprises a first column 301 for storing character codes (character managing numbers CD.sub.1 to CD.sub.n) representing codes corresponding to the character set table, not shown; a second column 310 for storing block start addresses BSADs which represent starting addresses in which bit map fonts corresponding to the character codes stored in the column 301, and a third column 320 representing memory capacities BQs of the bit map font data which correspond to the character codes stored in the areas of Table 300. The combinations thereof are as numerous as the number of characters, i.e. m. Accordingly, SAD.sub.1 of the font attribute address table 200 (FIG. 4) represents the address of CD.sub.1 while the EAD.sub.1 represents the address of BQ.sub.1.

Detailed Description Text (23):
FIG. 6 is a view showing a free block table.

Detailed Description Text (24):
An arrangement of the free block 400 of FIG. 3 will be described more in detail with reference to FIG. 6.

Detailed Description Text (25):
The free block table is in the table for storing the flag T indicating whether the blocks BK.sub.o to BK.sub.1 are used or not, wherein the blocks BK.sub.o to BK.sub.1 mean the blocks formed by dividing the bit map font area 136, the character attribute address table 300, the font attribute address table 200 and the free areas 134 and 135 by r bytes.

Detailed Description Text (28):
The bit map font area 136 is the area where the bit map fonts of all the pairs

composed of the font types TFs and the character sizes SZs of the font attribute address table are stored. Area 135 is divided into blocks BK.sub.o to BK.sub.p every r bytes. A block among the blocks, e.g. the block BK.sub.o stores in the starting address S.sub.o thereof the numbers of bytes to be actually used therein. A logical "0" is stored in an index register IX.sub.o (marked with reference numeral 511) located in the ending address of the block BK.sub.o when the bit map data corresponding to the character code CD can be stored in the block BK.sub.o. However, a starting address of the next block is stored in the index register IX.sub.o located in the ending address of the block BK.sub.o when the bit map data corresponding to the character code CD can not be stored in the block BK.sub.o. An arrow A shows that the index register indicates the starting address of the next block for storing the remainder of the bit map font data when the bit map data corresponding to the character code CD can not be stored in the block BK.sub.o. In FIG. 7, bit map font data are to be stored in area 512.

Detailed Description Text (32):
If the processor 4 decides that the font instructed to print by the host unit 20 has been previously stored in the font cache memory 6b, the processor 4 reads the bit map font data from the font cache memory 6b and writes the thus read bit map font data in the image memory 8 (S 105). Accordingly, the printing is carried out with use of the bit map font which was previously stored in the font cache memory 6b.

Detailed Description Text (33):
If the processor 4 decides that the font specified for printing by the host unit 20 is not stored in the font cache memory 6b, the processor 4 investigates whether the font memory car 13 is mounted on the printer or not (S 106). If the font memory card 13 is not mounted on the printer, a bit map font generating means generates the bit map font corresponding to the font which is instructed to print by the host unit 20 and writes the thus generated bit map font in the image memory 8 (S 107). Accordingly, the printing is carried out with use of the new generated bit map font.

Detailed Description Text (36):
If the processor 4 decides in decision diamond S 108 that the host-selected font is not stored in the font memory card 13, the bit map font data generating means generates a new bit map font data based on the information of the printing font supplies by the host unit 20 and writes the thus generated bit map font data in the image memory 8. At the same time, the decision means refers to the flag T in the free block table and decides as to whether the free block table is present or not in the memory space of the font memory card 13 (S 112).

Detailed Description Text (37):
The decision means decides the presence of the free block in the following manner in S 112. That is, the decision means retrieves successively the state of flag T.sub.n from the highest-order block BK.sub.1 of the free block table 400 to the lower-order blocks and decides that the block is free if the flag T.sub.n is a logical "0" and decides the free block is an area which is used by the font attribute address table 200. The flag T.sub.n corresponding to the free block is set to be a logical "1", thereby reserving the free block as the used area. Thereafter, the actual storage area is found by a predetermined calculation, thereby storing the data corresponding to the font attribute table 200 in the actual storage area.

Detailed Description Text (38):
If the processor 4 decides that the free block is not present in the memory space of the font memory car 13 in Step S112, processor 4 refers to a font managing table of the working memory 6 and decides whether the free area is present in the font cache memory 6b or not (S 113). If yes, then processor 4 writes the new generated bit map font data in the free area (S114), and writes the same bit map font data in the image memory 8 (S 115).

Detailed Description Text (39):
If the processor 4 decides that the free block table is present in the memory space of the font memory card 113 in Step 112, the processor 4 retrieves the free block from a free block table inspecting the state of the flag T therein and reserves the retrieved free block (S 116). The processor 4 stores the data of the font types and

the character sizes which are instructed to print by the host unit 20 in the free block reserved in the font attribute table 200. The processor 4 stores e.g. the logical "0" as an initial value in other items of the font attribute table 200 (S 117).

Detailed Description Text (40):
Thereafter, the processor 4 prepared data for each item of the character attribute address table 300 based on character data instructed to print by the host unit 20 and investigates the presence of a free block for storing the prepared data in each item of the character attribute address table 300 (S 118). The retrieval is carried out in the same manner as made in the Step 112, namely, by successively investigating the state of the flag T.sub.m from the highest order block BK.sub.1 to the lower order block of the block table 400 as shown in FIG. 6.

Detailed Description Text (41):
When the processor 4 retrieves the free block, it sets the flag T.sub.m corresponding to the free block to the logical value "1" and reserves the character attribute address table area (S 119).

Detailed Description Text (43):
Successively, the processor 4 retrieves a free area again in the free block table 300 by investigating the state of the flag T.sub.m (S 1221). If the free block is present, the processor 4 writes BSAD and BQ in the character attribute address table in the free block and stores the new generated bit map font data in the bit map font data block 512 corresponding to the bit map font area (S 122) and at the same time writes the same new generated bit map font data in the image memory 8 (S 123).

Detailed Description Text (44):
If the processor 4 decides in the Step 109 that the bit map font data of the character code which is instructed to print by the host unit 20 is not present in the character attribute table 300, the program goes to Step 118 where the processor 4 decides as to whether the free block is present in the memory space of the font memory card 13 by referring to the flag T. IF there is the free block, the program goes to Step 119 where the processor 4 stores the new generated bit map font into the free block table (S 112) and at the same time writes the new bit map font data in the image memory 8 (S 123).

Detailed Description Text (45):
If the processor 4 decides that the space area is not present in the font memory card 13 and in the font cache memory 6b by the judgement up to the Step 113, the processor 4 retrieves the font having the lowest frequency of use in the font memory card 13 using data 220 (FIG. 4), namely from the f's representing the frequency of use of the fonts in the font attribute table 200. The processor 4 erases the bit map font data and each item of the corresponding character attribute table 300 and the font attribute table 200 of the selected font having the lowest frequency of use and reserves the space area in the memory space of the font memory card 13 (S 125). Then, the program goes to Step 120 where the processor 4 stores each item data of the newly generated bit map font data character attribute table 300 and font attribute table 200 into the reserved free area so that the newly generated bit map font data is stored in the memory space of the font memory card 13 (S 122).

CLAIMS:

2. A printer provided with a font memory card according to claim 1, the printer having an input for receiving signals from a host control unit, the printer further comprising:

a font cache memory;

an image memory;

a first decision means for deciding as to whether a font designated by signals received at said input has been previously stored in said font cache memory of the printer;

a second decision means for deciding as to whether a font designated by signals received at said input has been previously stored in the font memory card;

characterized in that the bit map font data designated signals received at said input is written in said image memory when the first decision means decides that the designated font is stored in the font cache memory or the second decision means decides that the designated font is stored in the font memory card.

3. A printer provided with a font memory card according to claim 1, the printer having an input for receiving signals from a host control unit, the printer further comprising:

a font cache memory;

an image memory;

a first decision means for deciding as to whether a font designated by signals received at said input has been already stored in said font cache memory of the printer;

a second decision means for deciding as to whether a font designated by signals received at said input has been previously stored in the font memory card;

characterized in that the bit map font data having the least frequency of use among those stored in the font memory card is erased to reserve a free memory area and a new generated bit map font is stored in the free memory area reserved in the font memory card when the first decision means decides that the designated font is stored in the font cache memory and the second decision means decides that the designated font is not stored in the font memory card.

is coupled to the memory control unit 20 and then to channel 32. Virtual processors P.sub.0 -P.sub.7 are connected to the arithmetic unit 400 by means of a bus 402 with the arithmetic unit 400 communicating back to the virtual processors P.sub.0 -P.sub.7 by way of bus 403. The virtual processors P.sub.0 -P.sub.7 communicate with the internal bus 408 of the PPU22 by way of channels 410-417. A buffer unit, 419, having eight single word buffer registers 420-427 is provided. One register is exclusively assigned to each of the virtual processors P.sub.0 -P.sub.7. The virtual processors P.sub.0 -P.sub.7 are provided with a sequence control unit with the sequence control unit 418 in which implementation of the switch 401 of FIG. 3 is located. Control unit 418 is driven by clock pulses. The buffer unit 419 is controlled by a buffer control unit 428. The channel 429 extends from the internal bus 408 to the arithmetic unit 400.

Detailed Description Text (28):
The virtual processors P.sub.0 -P.sub.7 are provided with a fixed, read-only memory 430 known in the art. In the preferred embodiment of the invention, the read-only memory 430 is made up of pre-wired, diode array programs for rapid access.

Detailed Description Text (29):
A set of 64 communication registers 431 is provided for communicating between the bus 408, the I/O devices, and the data channels. In this embodiment of the system, the communication registers are provided in the unit 431.

Detailed Description Text (31):
The read-only memory 430 contains a pool of programs and is not accessed except by reference from the program counters of the virtual processors. The pool excludes a skeletal executive program and at least one control program for each I/O device connected to the system. The read-only memory 430 has an access time of 20 nanoseconds and provides 32-bit instructions to the virtual processors P.sub.0 -P.sub.7. Total program space in the read-only memory 430 is 1024 words. The memory is organized into 256 word modules so that portions of the programs can be modified without complete re-fabrication of the memory.

Detailed Description Text (32):
The I/O device programs may include control functions for the device storage medium as well as data transfer functions. Thus, motion of mechanical devices can be controlled directly by the program rather than by highly special purpose hardware for each device type. Variations to a basic program are provided by parameters supplied by the basic problem. Such parameters are carried in central memory units 12-19 or in the accumulator registers of the virtual processor executing the program.

Detailed Description Text (33):
The source of instructions for the virtual processors may be either read-only memory 430 or central memory modules 12-19. The memory being addressed from the program counter in a virtual processor is controlled by the addressing mode which can be modified by the branch instruction, or by clearing the system. Each virtual processor is placed in the read-only memory mode when the system is cleared.

Detailed Description Text (35):
Time slot zero may be assigned to one of the eight virtual processors by a switch on the control maintenance panel. This assignment cannot be controlled by the program. The remaining time slots are initially unassigned. Therefore, only the virtual processors selected by the maintenance panel switch operate at the outset. Furthermore, since program counters in each of P.sub.0 -P.sub.7 are initially cleared, selected virtual processor begins executing program from address zero of read-only memory 430 which contains a starter program typically known as a bootstrap. The selection switch on the maintenance panel also controls switch one of the eight bits in the file 431 is set by a bootstrap signal initiated by the operator.

Detailed Description Text (36):
The buffer 419 provides the virtual processors access to central memory modules 12-19. The buffer 419 consists of eight 32-bit data registers, eight 24-bit address registers and controls. Viewed by a single processor, the buffer 419 appears to be

only one memory data register and one memory address register. At any given time, the buffer 419 may contain up to eight memory requests, one for each virtual processor. These requests preferably are processed on a combined basis of fixed priority, and first-in-first-out priority. Preferably, four priority levels are established, and if two or more requests of equal priority are unprocessed, at any time, they are handled on a first-in, first-out basis.

Detailed Description Text (37):
When a request arrives at the buffer 419, it automatically has a priority assignment determined by the memory modules 12-19 arranged in accordance with virtual processor numbers and all requests from a particular processor receive the priority encoded in two bits of the priority file. The contents of the file are programmed by the executive program and the priority code assignment for each virtual processor is a function of the program to be executed. In addition to these two priority bits, a time tag may be employed to resolve cases of equal priority. The registers 431 are each of 32-bits. Each register is addressable from the virtual processors and can be read or written by the device to which it connects. The registers 431 provide the control and data links to all peripheral equipment including the system console. Some parameter switch control system functioning are also stored in the communication registers 431 from which control is exercised through the stored program controllers. FIG. 5: each cell in register 431 has two sets of inputs as shown in FIG. 5. One set is connected into the PPU22 and the other set is available for use by the peripheral device. Data from the PPU22 is always transferred into the cell in synchronism with the system clock. The gate for riding into the cell from the external device may be generated by the device interface, and not necessarily synchronously with the system clock.

Detailed Description Text (38):
FIG. 6. FIG. 6 illustrates structure which will permit allocation of a preponderance of the time available to one or more virtual processors P.sub.0 -P.sub.7 in preference to the others or to allocate equal time.

Detailed Description Text (39):
Control of the time slot allocation has between processors P.sub.0 -P.sub.7 is by means of two of the communication registers 431. Registers 431.sub.n and 431.sub.m are shown in FIG. 6. Each 32-bit register is divided into eight segments of four bits per segment. For example, the segment 440 of register 431.sub.n has four bits a-b which are connected to AND gates 441-444 respectively. The segment 445 has four bits a-b connected to AND gates 446-449 respectively. The first AND gates for all groups of four or the gates for all the "a" bits, namely AND gates 441 and 446, etcetra, are connected to one input of an OR gate 450. The gates for the "b" bits in each group are connected to OR gates to an OR gate 451 the third to OR gate 452, the fourth to OR gate 453.

Detailed Description Text (40):
The outputs of the OR gates 450-453 are connected to the register 454 whose output is applied to a decoder 455. Eight output decoder lines extend from the decoder 455 to control the inputs and the outputs of each of the virtual processors P.sub.0 -P.sub.7.

Detailed Description Text (42):
Three of the four bits 440, the bits b, c and d, specify one of the virtual processors P.sub.0 -P.sub.7 by a suitable state on the line of the output of decoder 455. The fourth bit, bit a, is employed to either enable or inhibit any decoding for a given set, depending upon the state of bit a, thereby permitting a given time slot to be unassigned.

Detailed Description Text (43):
It will be noted that the arithmetic unit 400 is coupled to the register 431.sub.n and 431.sub.m as by channels 472 whereby the arithmetic unit 400 under the control of the program, will provide the desired allocations in the registers 431.sub.n and 431.sub.m. In this response, thus in response to the clock pulses on line 460 the decoder 455 may be stepped on each clock pulse from one virtual processor to another depending on the contents of the register 431.sub.n and 431.sub.m the entire time may be devoted to one of the processors or may be divided equally or as unequally as

the codes in the registers 431.sub.n and 431.sub.m determine.

Detailed Description Text (44):
Turning now to the control lines leading from the output of the decoder 455, it is to be understood at this point that the logic leading from the registers 431.sub.n and 431.sub.m to the decoder have been illustrated at the bit level. In contract, the logic leading from the decoder 455 to the arithmetic unit 400 for control of the virtual processors P.sub.0 -P.sub.7 is shown, not at the bit level, but at the total communication level between the processors P.sub.0 -P.sub.7 and the arithmetic unit 400.

Detailed Description Text (46):
The flow of processor data on channels 478 is enabled for inhibited by states on lines 463-470. More particularly, channel 463 leads to an AND gate 490 which is also supplied by channel 478. An AND gate 500 is in the output channel of P.sub.0 and it is enabled by a state on line 473. Similarly, gates 491-497 and gates 501-507 control virtual processors P.sub.1 -P.sub.7.

Detailed Description Text (48):
FIG. 7 illustrates in block diagram, the interface circuitry between the PPU 22 and the CPU 34 to provide automatic context switching of the CPU while "looking ahead" in time in order to eliminate time consuming dialog between the PPU 22 and CPU 34. In operation, the CPU 34 executes user programs on a multi-program basis. The PPU 22 services requests by the programs being executed by the CPU 34 for input and output services. The PPU 22 also schedules the sequence of user programs operated upon by the CPU 34.

Detailed Description Text (51):
More particularly, a switch flag unit 44a will have enabled the switch 43a so that an indication of the next program to be executed is automatically fed via line 45a to the CPU 34. This enables the next program or program segment to be automatically picked up and executed by the CPU 34 without delay generally experienced by interrogation by the PPU 22 and a subsequent answer by the PPU 22 to the CPU 34. If, for some reason, the PPU 22 has not yet provided the next program description, the switch flag 44a will not have been set and the context switch would be inhibited. In this event, the user program within the CPU 34 that issued the SCW call would still be in the user processor but would be in an inactive state waiting for the context switching to occur. When context switching does occur, the switch flag 44a will reset.

Detailed Description Text (64):
It will be noted that the bus 32a and the bus 33a of FIG. 8 that the switching components responsive to the signals on lines 41a, 42a and 53-60 are physically located within and form an interface section of the PPU 22. The switching circuits include the OR gates 50a and 51. In addition, AND gates 61-67, AND gate 43a, and OR gate 68 are included. In addition, ten flip-flop storage units 71-75, 77-80 and 44a are included.

Detailed Description Text (75):
The salient characteristics of an interface between the CPU 34 and PPU 22 for accommodating the SCW and SCP and error context switching environment are:

Detailed Description Text (86):
Ten OR bits, i.e.: bits in one or more words in the communication register 431, FIG. 11, later to be described, are used for this interface. They are as follows in terms of the symbols shown in FIG. 4.

Detailed Description Text (89):
FIG. 12 shows the embodiment of the invention within an advanced scientific computer. The allocation of disc space on discs 38 and 39 is managed by the Disc Controller 704 which is a stored program controller located in Central Memory Stacks 12-19 wherein said controller executes within the PPU 22. The PPU 22 and other hardware processors access data from Central Memory Stacks 12-19 via the Memory Control Unit 20 which provides the necessary interface electronics. The Data Channel Controller 36 provides control for the moving of data between Memory Stacks 12-19

and the Disc Modules 38 and 39. The Disc Interface Unit 37 directs commands from the Data Channel Controller 36 to either Disc 38 or disc 39.

Detailed Description Text (92):
In FIG. 15, the Master Controller 601 is shown having access to the Central Processor 34, the Virtual Processors P.sub.0 -P.sub.7, the Printer 26, the Reader Punch 24 and the Read-Write 603. The Master Controller 601 will assign program units to the processors including the Central Processor and will perform initial interface with the external devices and monitor all processes.

Detailed Description Text (95):
The Task Controller schedules individual tasks and Central Processor steps within a command for execution. It allocates Central Memory. The Task Controller passes Disc I/O requests to the Disc I/O Controller. The Task Controller passes tasks and Central Processor programs to the Master Controller for proccesor assignment.

Detailed Description Text (97):
The Disc I/O Controller acts as the interface with the hardware Channel Controller. The Disc I/O Controller manages all disc requests so as to optimize effective channel bandwidth.

Detailed Description Text (99):
More particularly, FIG. 16 shows a block diagram of a part of the computer system of which the invention applies. Request for reading or writing on disc are generated by the control system of the computer of which the invention applies and are positioned into a High Priority Request Queue 700 or a Low Priority Request Queue 702, both of said queues being located in central memory stacks 12-19. The Disc Controller 704 is a stored controller also located in central memory 12-19 which executes within the pheripheral processor unit 22. The Disc Controller 704 operates upon requests in the two request queues 700 and 702, and always services all requests in the High Priority Request Queue 700 before those requests residing in the Low Priority Request Queue 702. All disc requests are operated upon by the Disc Controller 704 and placed in a variable length communication area (CA) Request Chain 705a, 705b, 705c, etc. The chain is a linked list of requests located in central memory 12-19 wherein the chain is constructed with the request ordered according to the physical areas on the discs 38-39 which the request addresses. In ordering the requests, Disc Controller 704 consults a 32-bit Disc Poisiton Word 706 in central memory which contains the present sector address of disc 38 in the first half of the word 706a and the sector address of disc 39 in the second half of the work 706b. Each disc module 38 and 39 contains a hardware register with the angular position address which is frequently read and placed in central memory location 706 by the control system. The Disc Controller 704 may place a new request between requests already existing on the Request Chain 705 when there is time to complete all requests in this order. In all cases where High Priority Requests and Low Priority Requests are vying for the same position in the chain 705, and when there is not sufficient time to service both requests, the low priority request is displaced in favor of the high priority request. The PPU 22 signals the Data Channel Controller 36 by means of changing the status of a bit in the Communication Register (CR) File 431. Once the Data Channel Controller is activated by the PPU 22, all requests on the CA Request Chain 705 are serviced without interruption. Each data link in the CA Request Chain 705 is composed of up to two octets of information which describe each request with such information as the address of the next octet in the chain, whether the request is a read or write, whether the request is for Disc Module 38 or 39, the address on said module, the address in central memory to or from which the data is to be transferred, and the number of words to transfer between the disc module and central memory.

Detailed Description Text (102):
The four stored program controllers which operate in different virtual processors within the computing system are the Master Controller, the Command Controller, the Task Controller, and the Disk I/O Controller. Each controller has one or more input and output queues which are configured in doubly linked-list fashion which provides for communication necessary between controllers. Typically, the output of one controller is positioned into the input queue of another controller.

Detailed Description Text (103):
Said Master Controller is at the top of the controller hierarchy and communicates with all processors and devices through an array of communication registers. Several of said registers are partitioned into bit fields which represent the request status of processors and devices which are monitored frequently by the Master Controller to determine when certain functions are to be performed. The Master Controller acknowledges requests by changing the bit status in the communication registers and passes the request to the next lowest level controller, the Command Controller.

Detailed Description Text (104):
The Master Controller has the ultimate scheduling authority for all hardware in the computer system and may force a processor to yield if a high priority resources requests service.

Detailed Description Text (106):
The execution of the command queue resulting from the operation of Command Controller is controlled by the next level of the control hierarchy, the Task Controller. To obtain the optimum efficiency of the virtual processors, it is desirable to time share their processing power among small program units. The commands in the queue constructed by the Command Controller are broken down into small program units called tasks. Each task has memory and execution time limits such that no system command will monopolize a processor in an unproductive manner. The Task Controller is the next in the controller hierarchy and has the function of scheduling each task, monitoring task linkage, managing transient system central memory, and insuring completion of commands as directed by the Command Controller. Task controller also has the responsibility of servicing requests for data transfers between memory and secondary and peripheral storage devices.

Detailed Description Text (107):
Task Requests for disc processing are the responsibility of the Disc Controller. The Task Controller constructs lists of disc input/output requests and the Disc Controller reorders said requests so that the use of the data channel associated with the discs is optimized.

Detailed Description Text (108):
The computer described herein is a multiprocessor. There are nine different processors in the computer with one central processing unit and eight virtual processors. There are thus in effect nine independent computers in the computing system. There are four control functions in the computer with a hierarchy of control. Four of the virtual processors are dedicated to the controller functions.

Detailed Description Text (111):
The Command Controller assigns groups of command sequences over groups of tasks to the computers. The Task Controller determines the task to be done such as intelligence gathering and the like. The Master Controller decides what processors will do the tasks. The Disc I/O Controller will control the transfer of data between the discs and central memory.

Detailed Description Text (114):
(1) provides software control of the virtual processor and central processor utilization,

Detailed Description Text (118):
There is a nucleus, which is the main polling loop of the Master Controller. The main polling loop of the Master Controller responds to device attention signals to provide a sufficient control of the virtual processors, central processor, the peripheral devices and terminals.

Detailed Description Text (119):
The major share of the Master Controller's work is initiated in a dedicated virtual processor to achieve a quick response time. This dedicated or nonselected virtual processor may be termed the Master Controller Virtual Processor and is one of the eight virtual processors other than a virtual processor which is allocated to the nucleus for the Master Controller.

Detailed Description Text (120):
VIRTUAL PROCESSOR UTILIZATION

Detailed Description Text (121):
The Master Controller performs four types of virtual processor utilization and control functions. These functions provide for:

Detailed Description Text (122):
(1) allocation of a virtual processor to a system task,

Detailed Description Text (123):
(2) allocation of a virtual processor for Master Controller work (subprocess).

Detailed Description Text (124):
(3) control of the virtual processors through a Master Controller service component (virtual processor service loop), and

Detailed Description Text (125):
(4) taking an active task out of virtual processor execution (trapping).

Detailed Description Text (126):
VIRTUAL PROCESSOR SERVICE LOOP

Detailed Description Text (127):
There is a virtual processor service loop component in the Master Controller which controls the nonselect virtual processes. The nonselect virtual processors enter and execute the Master service controller service loop upon completion of

Detailed Description Text (129):
(2) virtual processor execution for a task,

Detailed Description Text (132):
Control is also exerted over the nonselect virtual processors during Master Controller service loop execution to a yield indication provided in the communication register file. This is carried out through each logical break point in the virtual processor service loop execution, the nonselect virtual processors will test the yield indicator and will determine whether that nonselect virtual processor is to report to the Master Controller through the ROM idle loop. The yield indicators in the communication register file provide a means to interrupt a virtual processor during service loop execution so the Master Controller can use the nonselect virtual processor for special purposes.

Detailed Description Text (133):
The nonselect virtual processors report to a ROM idle loop when a nonselect virtual processor in the service loop detects a task processing and subprocessing scheduling lists are empty. The ROM idle loop indicates to the Master Controller that the nonselect virtual processors are idle and available for scheduling work.

Detailed Description Text (134):
TRAPPING VIRTUAL PROCESSORS

Detailed Description Text (135):
The Master Controller can exert control over a virtual processor during task execution by trapping (i.e., taking an active task out of virtual processor execution (a given virtual processor for use by the Master Controller). Trapping is accomplished through the setting of communication register file bits to which peripheral processor hardware responds, and the virtual processor traps to a ROM location where control is returned to Master Controller service loop.

Detailed Description Text (136):
Trapping is accomplished at the task level. All status information associated with the trapped disc is saved in the task parameter table and the task entry is placed at the front of the high priority task process list. Placement on the task process list allows a task to be rescheduled for execution in the next available virtual processor.

Detailed Description Text (138):
Subprocesses are initiated to perform systems services required at the Master Controller. By definition, a subprocess is a component of the Master Controller that performs a vital service and executes in a nonselect virtual processor. Subprocesses share a common data base with the nucleus of the Master Controller. Subprocess scheduling is a parameter of system generation in that some subprocesses may or may not be scheduled for nonselect virtual processor processing.

Detailed Description Text (139):
The Master Controller subprocess scheduler determines the order in which subprocesses are to be executed. The virtual processor service loop initiates subprocesses upon detection of entries on the subprocess scheduling list. The subprocess which are scheduled by the suprocess schedular are as follows:

Detailed Description Text (140):
(1) terminal input/output communication area processor,

Detailed Description Text (141):
(2) timer tube processor

Detailed Description Text (143):
(4) terminal input/output processor,

Detailed Description Text (144):
(5) performance information selective processor,

Detailed Description Text (145):
(6) device attention processor,

Detailed Description Text (146):
(7) central processor controller, and

Detailed Description Text (147):
(8) Master Controller communication list processor.

Detailed Description Text (148):
VIRTUAL PROCESSOR ALLOCATION FOR TASKS

Detailed Description Text (149):
System tasks are inputted to the Master Controller on one of two priority task priority processing lists for virtual processor allocation. When a task is selected for virtual processor execution, the task entry is removed from the task processing list and is then primed for execution. Prior to task execution, the Master Controller builds table entries which point to an active tasks task entry and task parameter table, and loads a base program counter and a virtual process register file. The base program counter and the virtual processor register file are both contained in the task processor table. The control of the virtual processor is then passed to the task for execution.

Detailed Description Text (150):
The time a task is permitted to execute in the virtual processor can be controlled by the Master Controller. A virtual processor which enters an endless program loop in execution will be discovered by the Master Controller and will be removed from virtual processor execution. The Master Controller also services requests from task execution in the virtual processor for resetting the communication register file or the central memory protection when required. A task is post-processed by the Master Controller upon completion of virtual processor execution. This post-processing may be termined a task wrap-up.

Detailed Description Text (151):
Following the task execution, the base program counter and the virtual process registers for that task are saved in the task parameter table. The task parameter table is removed from the task parameter table address list and the task entry is removed from the active task list. The task entry is placed on the task complete

queue for the task controller's wrap-up of the task, and control is returned to the virtual processor service loop.

Detailed Description Text (152):
CENTRAL PROCESSOR UTILIZATION

Detailed Description Text (153):
The central processor utilization is accomplished by the Master Controller through

Detailed Description Text (154):
(1) allocation of the central processor to central processor steps,

Detailed Description Text (155):
(2) responding to serve at central processor service calls, and

Detailed Description Text (156):
(3) control of the central processor.

Detailed Description Text (157):
The Master Controller exerts control over the central processor hardware logic that is necessary to place a central processor program into execution. Allocation is accomplished by removing entries from the central processor execution list and placing the address of the program into a dedicated word which causes the hardware to start execution of the program.

Detailed Description Text (158):
The Master Controller services the central processor for context switching issued by a central processor step through a monitor call and wait instruction. Context switching is the hardware mechanism for removing the central processor from execution and saving the status associated with the central processor step.

Detailed Description Text (159):
A central processor step will issue a service call (the step may or may not be context switched) that signals the Master Controller through the communication registers that the central processor step requires system services. The Master Controller intercepts the service call and builds a service request entry for passing the necessary information to the command controller for processing.

Detailed Description Text (160):
Context switches of the central processor may be forced by the Master Controller when a high priority step needs to be assigned to immediate central processor execution. Priority steps may also be forced from execution when they have exceeded a given time limit or for the purpose of time slicing the central processor. The central processor step is normally context switched automatically by the hardware for step termination. However, when there are no central processor steps waiting or trying for central processor execution, the Master Controller forces a central processor step out of execution through use of the central processor maintenance hardware logic to terminate a step. The Master Controller also forces the central processor steps into execution through use of the central processor maintenance hardware logic.

Detailed Description Text (165):
The Master Controller interfaces with the peripheral devices, peripheral processor, central processor, and the terminal channel controllers to the central register file. Information may be inputted from new devices and control signals may be outputted through these devices from the communication register file.

Detailed Description Text (174):
The master controller provides special functions for master control debug. Special control of the virtual processors is permitted by an exercise task which completes execution to stop all other exercise virtual processors at that time. The master control debug users can then examine the relationship between one or more virtual processors in simultaneous execution.

Detailed Description Text (177):

The master controller also provides a special interface for initiating the execution of the performance information collection central processing step that empties and processes operating system data collection buffers.

Detailed Description Text (179):
The nucleus of the master controller executes at a selected virtual processor that is composed of three main components. These components are:

Detailed Description Text (182):
(3) the peripheral processor controller.

Detailed Description Text (183):
These three components share various tables and have a common data base structure. The nucleus of the master controller monitors the communication register file, in order to communicate with the virtual processors, the central processor, the peripheral devices and the terminal channel.

Detailed Description Text (185):
The main polling loop of the master controller monitors requests from the external world. Whenever a signal is detected for master controller work, the main polling loop initiates the appropriate action. To initiate the request, the main polling loop will pass control to the peripheral processor controller or the subprocess scheduler in the selected virtual processor.

Detailed Description Text (186):
When the main polling loop detects that the execution of a subprocess is required, control is passed to the subprocess scheduler. The subprocess scheduler may pass control to the requested subprocess if the master controller virtual processor select scheduling is required. The scheduling is time dependent, then subprocessor schedular passes control to the peripheral processor controller for immediate virtual processor allocation. If the subprocess is not time dependent and there is no available virtual processor, than the subprocess schedular makes an entry in a table so the subprocess qwill be executed by the next available virtual processor. After the scheduling of a subprocess, control is returned to the main polling loop of the master controller.

Detailed Description Text (187):
Control of virtual processor select is passed to the peripheral processor controller whenever the slave to master or availability communication register bits are set. These bits indicate that a virtual processor needs servicing by the peripheral processor controller. The reason word indicates to the peripheral process controller the state of the virtual processor. Following servicing, control is returned to the main polling loop of the master controller.

Detailed Description Text (190):
The central processor is monitored for:

Detailed Description Text (198):
The peripheral processors are monitored for virtual processor availability and system error detection.

Detailed Description Text (211):
The master controller debug component is monitored by special virtual processor control requests.

Detailed Description Text (215):
(1) virtual processor-select only,

Detailed Description Text (216):
(2) available virtual processor, a virtual processor-select immediately,

Detailed Description Text (217):
(3) available virtual processor, or trapped virtual processor,

Detailed Description Text (218):

(4) next virtual processor becomes available.

Detailed Description Text (219):
Subprocesses which are scheduled for execution into a virtual processor other than the master controller virtual processor-select may be queued up while waiting for a virtual processor to become available.

Detailed Description Text (220):
PERIPHERAL PROCESSOR CONTROLLER

Detailed Description Text (221):
The function of the peripheral processor controller is to service requests associated with the peripheral processor. These include:

Detailed Description Text (224):
(3) available virtual processor scheduling, and

Detailed Description Text (225):
(4) master controller debug virtual processor control.

Detailed Description Text (226):
These requests are signalled from nonselect virtual processors through communication register bits assigned to each virtual processor and an octet of dedicated central memory. A virtual processor requiring servicing sends its communication register bit, called slave-to-master bit, indicating that it has stored a slave-to-master reason in the dedicated central memory octet. This reason in the dedicated central memory octet has been accessed by the peripheral processor controller when the appropriate action is taken.

Detailed Description Text (254):
When a command is scheduled, the command entry for that command is placed on the command processing list. A task entry is placed on the task request queue by command controller for the first task in the command to be initiated by task controller.

Detailed Description Text (258):
(1) the internal job specification language analyzer which is a part of the command controller request command sequences to processs the work specified on job specification language statements. The master controller receives service calls from the central processor and passes the requests to the command controller. The command controller then initiates the command sequences needed to analyze the requests. The peripheral processor debug, which is the on-line operating system debugging system, generates requests for command sequences to handle operation interaction with the peripheral processor and the central processor debug facilities. The system communicator operates requests for command sequences to handle operator interaction, to cancel jobs, to delete input/output requests and the like. The command sequences are also used for the analysis and processing of terminal requests by the command controller.

Detailed Description Text (261):
The command controller subprograms are functionally similar to tasks since a command controller subprogram represents a certain unit of code or a certain sequence of instruction. However, the command controller subprograms are actually exeucted in the command controller's dedicated virtual processor. Because the command controller comprehends commands as units of work, the command controller subprograms have command level data structures. That is, the command controller subprograms are activated by an entry in the command sequence table and have a command entry and a command parameter table. The command controller subprograms perform different functions for the control system.

Detailed Description Text (263):
The second type of subprogram manipulates lists within the command controller. Command controller subprograms which manipulate lists solve a lockout problem. More than one component accesses a list to add or remove list entries a lockout problem may exist. The command controller subprograms resolve this type of problem since only one subprogram will be executed in the command controller's dedicated virtual

processor.

Detailed Description Text (264):
The command controller subprograms are also used to resolve interface incompatibility that has occurred because of miscommunication or unforeseen problems in the design phase. For example, a command may need parameters not available to the calling components. The command controller subprogram can obtain and supply these parameters which will be necessary for that command to execute.

Detailed Description Text (265):
The command controller subprograms may also be used because of size. Where the code requires less than 20 instructions, it becomes reasonably economical to execute these instructions in the command controller's dedicated virtual processor. This type of subprogram cannot be exectuted as a last task of a command because if the command is context independent it may thus be a command in other command sequences which do not require execution of the subprogram.

Detailed Description Text (268):
The command controller multiplexes two functional units within one virtual processor as shown in FIG. 10. These two functional units are the scheduler component of the command controller and the polling loop subprogram executer component and command controller. This design of the command controller allows the scheduling components to be altered without impacting any of the bookkeeping and commands sequence interpretation functions of the command controller.

Detailed Description Text (270):
Both the scheduler and the polling loop components are divided into subcomponents called actions. An action is a modulo module of code which performs a specialized function in the command controller. Actions are implemented to provide a modular design for the command controller. The sequence in which these actions are executed is determined by an action sequence table in the command controller's task parameter table. The action sequence table contains an entry for each action in each entry defines two possible exits for that action, thus determining which action will be performed next. There are two action sequence tables: one for one scheduler and one for the polling loop and subprogram executor.

Detailed Description Text (271):
After each action control is returned to the multiplexing driver. This driver decides which of the two components of the command controller will execute next. This is accomplished by checking in ratio (scheduler-executor ratio) built into the command controller's task parameter table at system generation time. This ratio determines which component will be allowed to execute more frequently. For example, the scheduler could be allowed to execute one hundred actions for every five actions that the polling loop executes. The multiplexing driver must keep a running count of the number of actions executed by each component and initiate succeeding actions accordingly. When an action terminates, an action sequence table interpretor determines which action for this component should be executed next and stores the address of that action in the dedicated word of the command controller's task parameter table. Control is transferred to the multiplexing driver which determines which component will execute next and transfers control directly to the correct action of that component.

Detailed Description Text (272):
The command controller has the same basic data structure as a task, that is, a task entry, a task parameter table, a command entry and a command parameter table under a system job. The command controller has base relative access to its own task parameter table and stores information in it. Temporary storage is provided for command controller execution, and data constant areas are provided for the different components of command controller. Within the data constant areas, there are counters which control the execution flow within command controller by indicating to the various components how many times queues should be scanned and how many entries should be removed from a queue each time it is scanned.

Detailed Description Text (273):
Command controller's task parameter table contains the command directory of all the

commands in the system. Each entry in the command directory contains information such as:

Detailed Description Text (282):
Also, in the command controller's task parameter table, there is a command sequence table directory which contains the addresses of all the command sequence tables, the address of the first command in each command sequence, and other information similar to that in the command directory. The action sequence tables are also in command controller's task parameter table and are used to control the flow of work inside the command controller.

Detailed Description Text (284):
The task controller is the component of the ASC Operating System responsible for monitoring the flow of tasks through the system. A task is the most basic unit of work scheduled by the sytem. Task controller is central memory resident and executes in a dedicated virtual processor.

Detailed Description Text (286):
(1) allocates central memory for executable code and builds parameter tables needed for the execution of a task,

Detailed Description Text (290):
(5) provides the interface between the system and the peripheral device driver/handlers.

Detailed Description Text (295):
In preprocessing tasks, task controller loads tasks from disc to central memory if the tasks are not currently residing in central memory. After tasks have been loaded into central memory, task controller builds a task parameter table to satisfy the task data structure requirements for task execution. Task controller also records the current status of a task which involves the setting of a task which involves the setting of various indicators contained in the task directory. Task execution is initiated by placing the task entry on a processing list to master controller for virtual processor allocation.

Detailed Description Text (296):
Task controller also services tasks which has completed execution. Task completions result from the execution of a standard call (SCALL). A SCALL is a standard system linkage to initiate a subsequent task or pass control to a system control component. For a task-to-task call, task controller modifies the activation record of the terminating task to reflect initiation of the requested task and performs the preprocessing required to initiate the task being called. If a task's SCALL request requires command controller attention, task controller places the task entry of the calling task on the proper action list for command controller to process. For a SCALL requesting disc input/output services, task controller links the disc input/output request on a list for disc preprocessing. For SCALLs to terminal input/output and to drive manager, the appropriate list entry, previously built by the calling task, is placed on the proper list for processing.

Detailed Description Text (298):
The task directory is a control component system table which contains an entry for each task in the system. Each task directory entry contains all the information required by task controller to preprocess the task. Whenever the task directory is accessed, the entry which is used is locked out from other system components which might try to access the same entry.

Detailed Description Text (299):
The task directory entry for each task specifies whether the task is currently resident in central memory or whether it is to be loaded from disc into central memory. Certain tasks will always be central memory resident, but most tasks will reside in central memory only during the execution of the task code. The task directory also indicates whether the task is serially reusable or reentrant.

Detailed Description Text (303):
The task directory indicates to task controller the amount of central memory

required by a task during execution for the task parameter table. Before the central memory is reserved, task controller sugments the task parameter table size to include all overhead words required by the system. Another task directory parameter indicates the actual code length, including all data constants, required for a particular task. The task code length is required by task controller if a disc input/output request is required to load the task from disc. A data displacement parameter in the task directory indicates the displacement from the load point of a task to the data constants associated with the task.

Detailed Description Text (304):
A central memory address is contained in the task directory to indicate the load point of a task when the task is in central memory. For code being loaded by a disc input/output request, task controller updates this central memory address to indicate the current position in central memory of the task code. There is also a word in each task directory entry to indicate the absolute disc address of tasks which reside on disc.

Detailed Description Text (313):
The input lists to task controller consist of the following:

Detailed Description Text (314):
(1) task request lists, on which command controller and task controller place task activation records for tasks to be executed;

Detailed Description Text (315):
(2) task complete lists, on which master controller places task entries of tasks having executed a SCALL;

Detailed Description Text (317):
(4) memory deallocation list, on whih command controller and task controller place task entries and task parameter tables for task controller to deallocate later;

Detailed Description Text (318):
(5) large memory request list, on which command controller places task entries for task requests requiring central memory at the page (4096 words) level;

Detailed Description Text (320):
(7) breakpoint complete list, on which master controller places task entries of tasks having executed a software breakpoint;

Detailed Description Text (321):
(8) breakpoint restart list, on which command controller places task entries of tasks to be restarted after executing a software breakpoint.

Detailed Description Text (322):
The priorities of the various input lists are satisfied by the multiple occurrence of a list or lists in the polling loop. Once an event is detected in the polling loop, task controller exits the loop to the particular processor required and returns to the polling loop when all processing is completed.

Detailed Description Text (324):
Driver manager is that part of task controller which provides the interface with the peripheral input/output driver/handlers to effect data transfers between central memory and the peripheral devices (e.g., tape, card reader, punch, printer).

Detailed Description Text (325):
When peripheral I/O needs to be done for a task, the task will pass control to task controller via a special SCALL. After the task controller recognizes from the SCALL type that peripheral input/output activity needs to be initiated, control will be passed to a part of task controller called driver manager. Driver manager will initiate the peripheral input/output activity as indicated by an argument table passed to task controller by the calling task.

Detailed Description Text (326):
Driver manager will pass control to a device driver/handler component to initiate

the data transfer. When the data transfer has been completed, control will be returned to the driver manager portion of task controller by the driver/handler. Driver manager will then pass control to the command controller, which will continue to process the command sequence that requested the peripheral input/output.

Detailed Description Text (327):
FIG. 11 and outline description illustrate the interfaces involved for driver manager from a peripheral I/O request which was initiated to task controller.

Detailed Description Text (329):
(b) task controller nucleus passes control and the peripheral I/O request to the driver manager SCALL processor.

Detailed Description Text (330):
(c) if the device handler is in central memory, the driver manager SCALL processor transfers the data from the peripheral I/O request into a dedicated communication area and sets a communication register bit which causes a driver to initiate a device handler. Driver manager SCALL processor may make an entry on a task processing list for a driver if the driver is not executing in a virtual processor.

Detailed Description Text (331):
(d) if the device handler is not in central memory, driver manager SCALL processor will make a disc I/O request for the handler to be loaded from disc.

Detailed Description Text (332):
(e) when the disc I/O request is completed, control is given to the post handler load processor.

Detailed Description Text (333):
(f) the post handler load processor does the same processing that is described for the driver manager SCALL processor under point (c) above.

Detailed Description Text (335):
(h) the device handlers interface directly with the hardware to perform data transfers between the peripheral I/O devices and central memory.

Detailed Description Text (336):
(i) the master controller nucleus passes control to the buffer attention processor. The buffer attention processor puts a disc I/O request on a list which will activate disc I/O preprocessing in task controller. The disc I/O request is supplied by the task whose peripheral I/O request is being serviced.

Detailed Description Text (338):
(k) the task controller nucleus will pass control to the driver attention processor.

Detailed Description Text (339):
(l) the driver attention processor informs command controller of the completion of the peripheral I/O request.

Detailed Description Text (343):
Task Controller Disc Preprocessor is a list driven component. A list entry is placed on one of three disc I/O requests lists by task controller before task controller disc preprocessor is called. Task controller calls TCDP as a subroutine through the vector table. There are three priority disc I/O requests lists, the highest priority being used for one inch streaming tapes. Althrough the list entry is the only form of input for TCDP, the list entry contains pointers to control blocks. These control blocks contain the information necessary to service a disc I/O request.

Detailed Description Text (351):
Communication areas (CAs) are allocated and built by TCDP upon completion of request validation and absolute disc address calculations. The communication areas are used by the disc hardware to obtain the information that is required for the actual disc data transfers. After being built, a communication area is linked on one of two lists which initiates disc controller processing. Task controller disc preprocessor

returns to the task controller when an error condition occurs or all communication areas have been built for a given request control block. When all the communication areas have been built for a disc I/O request (more than one request control block may compose a disc I/O request), the disc request list entry is removed from the task controller disc preprocessor input list and placed on an "in process" list.

Detailed Description Text (357):
TASK CONTROLLER LIST PROCESSING

Detailed Description Text (358):
A major portion of task controller's function is accomplished through response to inputs on linked lists which task controller polls. The main polling loop of task controller monitors the lists associated with task controller processing and initiates activity to process the list entries.

Detailed Description Text (359):
To initiate a task, command controller places an entry on the task request list. When task controller finds the entry on the task request list, the task to be started is preprocessed and initiated by task controller.

Detailed Description Text (360):
Another element on task controller's polling loop is the large memory request list. Inputs on the large memory request list are similar to inputs found on the task request list in that entries on the large memory request list are task entries for the initiation of tasks. The difference is that tasks being initiated through the large memory request list require central memory usage at the page (4K words) level. Tasks such as the central processor step loader and tasks requiring peripheral I/O transfers with large central memory buffers are placed on the large memory request list. Task controller processes the large memory request list by searching the list, determining the memory requirements, and communicating with page management (a component of memory management) to determine if the central memory requirements are available. The required amount of memory is reserved when page management can satisfy the memory needs of a task entry on the large memory request list. After satisfying the memory requirements, the large memory request entry is processed in the same manner as entries from the task request list.

Detailed Description Text (361):
Task controller polls a deallocation list to determine if any task activation records are to be deallocated. The deallocation list may contain a task parameter table (TPT) or a task entry (TE). Task controller deallocates the task entry, the task parameter table and all blocks of central memory associated with the the task.

Detailed Description Text (363):
Task controller polls the low priority terminal I/O list to determine when a terminal I/O request is complete. If the list entry indicates that the request is complete, task controller delinks the entry and deallocates the memory for that entry. If a disc I/O request is required to complete a terminal-to-disc transfer, task controller is responsible for initiating the disc input/output request. If a disc I/O request has completed and a central memory-to-terminal transfer is required, task controller initiates the transfer. Task controller also allocates and deallocates all central memory required for terminal-to-disc and disc-to-terminal transfers.

Detailed Description Text (364):
Master controller places task entries on the task complete list for SCALLs being processed in the task's virtual processor. For standard task-to-task communication, task controller completes the wrapup of the current task and preprocesses the task being requested. For disc I/O or driver manager SCALLs, task controller is responsible for passing an argument list to the particular I/O preprocessor. For terminal I/O calls, task controller passes an argument table to the terminal I/O preprocessor. Also for terminal I/O calls, task controller is responsible for initiation data transfers from disc to terminal, terminal to disc, and for providing the central memory necessary for these transfers. For central processor execution requests, task controller places the argument list passed (a step entry) onto the central processor execution list. All other SCALL types are treated as exits to

command controller. Task controller places a status and function code in a command entry and places this entry on command controller's command attention list.

Detailed Description Text (366):
Software breakpoints in the operating system interface with the systems analyst via the master controller debug component (MCD). Prior to invoking the master controller debug component, however, all levels of control are involved in breakpoint servicing.

Detailed Description Text (367):
When a task hits a breakpoint, master controller places the task entry (TE) on a breakpoint complete list to task controller. If the breakpoint was to interrupt other executing tasks, their task entries are also placed on the breakpoint complete list. The task entry and task parameter table of the task retain sufficient task status information to allow the restart of the task. Task controller's functions at breakpoint time are to

Detailed Description Text (368):
(1) gather all task entries on the breakpoint complete list associated with one breakpoint into an argument list,

Detailed Description Text (370):
(3) invoke the master controller debug component command sequence which services the argument list of task entries.

Detailed Description Text (371):
Upon the master controller debug component's completion of breakpoint servicing, command controller places the command entries back on the command processing list and task controller receives the task entries. If the task entry indicates the breakpoint halted this task during execution of the task code proper, the task is placed on the front of the high priority task processing list for restarting by master controller. If the breakpoint in another task caused this task to be terminated during the execution of a SCALL, the SCALL was allowed to complete and task controller places this task entry on the task complete list for SCALL servicing.

Detailed Description Text (376):
The task controller services for exercise tasks consist of maintaining an exercise task directory for those temporary tasks entered into the operating system by the master controller debug component user. The exercise task directory is a variable-length linked list of entries supplied by the master controller debug component. Each entry contains the exercise task's identification number as well as all of the information as specified for each system task in the system task directory, with the one exception being the disc address since exercise tasks will be central memory resident for the duration of their existence. Task controller accepts exercise task directory entries from the master controller debug, validates them with respect to their being central memory resident, in central memory flags, and enters them in the exercise task directory. Task controller also deletes entries from this directory upon request of the master controller debug component.

Detailed Description Text (388):
Each I/O component is relatively independent; i.e., the interfaces between them are obtained through standard system linkages and control blocks.

Detailed Description Text (389):
The logical I/O central processor subroutine provides central processor (CP) users with a logical record file concept and removes the responsibility for physical block formatting of disc records from disc I/O. Fortran and assembly language programs use logical I/O to process their disc files. Logial I/O processing provides the user the capability to transfer records between disc and central memory (CM), therefore relieving the user of responsibility of making physical disc I/O requests.

Detailed Description Text (390):
Fortran I/O is a central processor subprogram that provides an interface component between the Fortran program and logical I/O. Fortran I/O is used only during

execution of a program, and a separate copy of the required components of Fortran I/O is loaded with each central processor program.

Detailed Description Text (392):
Disc management also provides the user a standard set of system procedures for defining a disc I/O request. These procedures are used by the central processor user to allocate control block space and to define the type of disc I/O request. The request control block (RCB) shown in FIG. 14 contains the control parameters necessary for executing a disc request. A facility is also available to chain 705 or stack 700, 702 requests, but only one call is issued the system for serving the request. The system accepts the disc requests and passes them to disc input/output (DIO) for processing. Upon completion of the disc request the associated status is recorded in the request control block. A system service that tests for the completion of the disc request is provided to be called by the central processor program.

Detailed Description Text (404):
The above mentioned components are serially reusable and are central memory resistant code. Task controller disc preprocessor and task controller execute in the same virtual processor. Disc controller executes in its own dedicated virtual processor.

Detailed Description Text (407):
(b) task controller links the disc request on a list and passes control to the task controller disc preprocessor (TCDP).

Detailed Description Text (418):
TCDP is a list driven component. A list entry is placed on one of three disc I/O request lists by task controller before TCDP is called. Task controller calls TCDP as a subroutine through the vector table. There are three priority disc I/O request lists, the highest priority being used for one inch streaming tapes. Although the list entry is the only form of input for TCDP, the list entry contains pointers to control blocks. These control blocks contain the information necessary to service a disc I/O request.

Detailed Description Text (431):
(2) post request processor, and

Detailed Description Text (436):
(2) completion of disc I/O requests (post request processor), and

Detailed Description Text (442):
Any severe system error conditions that are detected will be processed first and cause the disc controller to stop processing. When a parity error is detected for a completed communication area, priority scheduler will reschedule the communication area so the error may be corrected. Part of priority scheduler's work is to determine if any CAs have been completed by the disc hardware. If a high priority CA on a disc channel list has been completed, then priority scheduler will schedule this communication area (CA) for postprocessing before entries on the CA input lists. There are two priority CA input lists in which entries are supplied by task controller disc preprocessor. The priority scheduler will schedule the highest priority list entries first for processing. In summary, priority scheduler provides the analysis necesary to determine what action is to take place next in disc controller.

Detailed Description Text (451):
Post Request Processor

Detailed Description Text (452):
The post request processor is initiated by the priority scheduler when the decision has been made to process a completed communication area from one of the disc channel lists. The post request processor processes completed communication areas by:

Detailed Description Text (458):
Post request processor is primarily responsible for completion of the total disc I/O

request. In the majority of disc I/O requests, there will be associated several communications areas. Since the scheduled communications areas for a given disc request will complete at different points in time, there must be provided a method to record communications areas completions. FIG. 14 illustrates how post request processor controls the completion of a disc request.

Detailed Description Text (459):
The arrows (PTR) illustrate the linkages provided for post request processor to access the different control blocks associated with a disc request.

Detailed Description Text (465):
Upon completion of a disc I/O request, post request processor passes the completed request to task controller and returns control to priority scheduler.

Detailed Description Text (466):
Having described the control system for the computer of which the invention forms a part, the stoed program processors of the invention are now described.

Detailed Description Text (483):
.5 Interfaces:

Detailed Description Text (583):
E. WCP-Waiting Command Processor

Detailed Description Text (584):
F. SEP-SOFT Expansion Processor

Detailed Description Text (633):
B. If PIOFIB exists and shared use is not allowed; the registers are saved as parameters to call Open's task WCP (Waiting Command Processor).

Detailed Description Text (670):
.5 Interfaces

Detailed Description Text (743):
.1 Pre-processor--Its function is to validate an allocation request, select the channels/modules required to carry out the allocation, set up the Task Parameter Table (TPTB) to be used by future tasks in DAC, and interface with the allocation and release processes.

Detailed Description Text (746):
.6 Tables--The following tables are used by the tasks of DAC:

Detailed Description Text (766):
.1 The Job Scheduler component determines the amount of disc space resources reserved for a job. The allocation processor of DAC performs the actual assignment of the disc space reserved to a job among its various files.

Detailed Description Text (773):
DISC PRE-PROCESSOR

Detailed Description Text (774):
1. Purpose--Disc Assignment Pre-Processor (DPP) is responsible for the validation of request for disc space. If the validation is successful it also initializes certain quantities in the Task Parameter Table (TPTB) for use by the following Disc Assignment tasks.

Detailed Description Text (775):
2. Input--Input to DPP is a parameter word in the Task to Task Parameter Table (TTPT) consisting of a function code and a parameter block pointer as shown in Table 10

Detailed Description Text (792):
Flow charts for the Disc Assignment Pre-Processor are shown in FIGS. DA2-DA4.

Detailed Description Text (795):
.2 Input--Input to DMS is the Task Parameter Table (TPTB) used by the Disc Pre-Processor routine. Any input information required by this task will have been placed in the TPTB by DPP.

Detailed Description Text (797):
.1 Disc Assignment Processor--If there is enough free space (not assigned to either active or inactive files) DAP will be the task to execute after DMS. DMS furnishes this task with the order for processing modules in disc assignment and the amount of disc which is to be assigned to the various modules. These items are stored in the TPTB which is to be passed on to DAP.

Detailed Description Text (805):
DISC ALLIGNMENT PROCESSOR

Detailed Description Text (806):
.1 Purpose--Disc Assignment Processor (DAP) performs the actual assignment of the disc space reserved to a job among its various files. It assigns the actual disc space necessary to satisfy the user request and builds a map of the allocated disc space.

Detailed Description Text (807):
.2 Input--DAP will be entered from either of two tasks; DMS if it was determined that there was enough space available for assignment without releasing inactive files or DRP if it was necessary to release inactive files in order to provide the space necessary for assignment. Any input information required by this task will have been placed in the task parameter table by DRP or DMS.

Detailed Description Text (808):
.3 Output--Output from Disc Assignment Processor will depend upon the type of completion.

Detailed Description Text (814):
.1 If a fragment of the desired size is not available for assignment the Disc Assignment Processor will be able to either borrow a fragment from a domain of larger fragments and use a portion of it or assign several smaller fragments until the desired amount of disc space have been assigned.

Detailed Description Text (819):
Flowcharts for the Disc Assignment Processor routine are shown in FIGS. DA10-DA19.

Detailed Description Text (821):
.1 Purpose--Disc File Release (DFR) acts as a controlling task for Disc Release Processor. It is responsible for removing the information pertinent to a file prior to the release of the file. It also decides which files are to be released immediately and which are to be placed on the Inactive File Chain.

Detailed Description Text (822):
.2 Input--DFR will be started when the Disc Assignment Processor determines that there is not enough free space available to fill a request, making it necessary to release files on the Inactive File Chain or when Disc Pre-Processor determines that the Disc Assignment Component has been called to release a file. The two word input parameter block located in the TPTB will be as shown in Table 11.

Detailed Description Text (832):
.1 Disc File Release divides files into three major groups: catalog files, catalogs, and other files. In order to keep catalog files on disc for as long a time as possible they will be placed on an inactive file chain until the space which they use is required by another file. They will not be released until Disc Assignment Processor determines that the disc they occupy is required.

Detailed Description Text (837):
PROCESSOR

Detailed Description Text (838):

.1 Purpose--Disc Release Processor (DRP) processes the disc map built for a file by the Disc Assignment Processor and updates the DAC tables making space defined by the map available for reassignment. DRP also searaches the Bad Band Table (BAND) to determine if any of the bands on which the released file resided contained read errors. Any bands so noted are marked unavailable for reassignment.

Detailed Description Text (839):
.2 Input--DRP will be entered from the Disc File Release (DFR) task. Any input information required by this task will have been placed in the task parameter table by DFR.

Detailed Description Text (840):
.3 Output--There are no error returns within this task. Upon completion this task will return to DFR with the Disc Assignment Component tables updated. It will also return to DFR the amount of space released in band or larger fragments within the updated Task Parameter Table.

Detailed Description Text (842):
Flowcharts for the Disc Release Processor are shown in FIGS. DA26-DA32.

WEST

Generate Collection Print

Just 7 months earlier

L25: Entry 4 of 34 File: USPT Jul 29, 2003

DOCUMENT-IDENTIFIER: US 6601153 B1
TITLE: Method and apparatus for increasing computer performance through asynchronous memory block initialization

Abstract Text (1):
A system and method for increasing processing performance in a computer system by asynchronously performing system activities that do not conflict with normal instruction processing, during inactive memory access periods. The computer system includes at least one instruction processor to process instructions of an instruction stream, and a memory to store data. One or more inactive data blocks in the memory are identified, and a list of addresses corresponding to the identified inactive data blocks is generated. Available computing cycles occurring during processing in the computer system are identified, such as processing stalls and idle memory write periods. The inactive data blocks associated with the list of addresses are initialized to a predetermined state, during the available computing cycles. Addresses corresponding to those initialized data blocks are then made available to the computing system to facilitate use of the data blocks.

Detailed Description Text (5):
Similarly, systems employing cache memories also provide information in blocks. Data stored in cache memory is more quickly accessible than data stored in a main memory area such as RAM. When, for example, a read function for a particular data segment is attempted, the cache (or multiple caches in the case of multi-level caching) is inspected first to determine whether the requested data is in the cache. If so, it is a "hit", and the data is read from the cache. If the data is not in the cache, it is a "miss", and the data must be retrieved from RAM. However, data is not retrieved piece-by-piece from the RAM, but rather is typically retrieved in "cache lines" or "cache blocks". Generally, a cache line is the smallest unit of memory that is transferred between the main memory and the cache. Each cache line or block includes a certain number of bytes or words, and a whole cache line is read and cached at once. This takes advantage of the principle of locality of reference, which stands for the proposition that if one location is read, then nearby locations (particularly following locations) are likely to be read soon thereafter. It can also take advantage of page-mode DRAM, which allows faster access to consecutive locations.

Detailed Description Text (31):
If the processor does not already have a page address stored in a register, the processor takes the next page address from the to-be-zeroed queue 224 as seen as operation 308. The page associated with that start address is the page that the processor will operate on until the page is successfully and fully cleared. This start address is preferably stored in an internal working register of the processor, such as the internal working register discussed in connection with operation 306. If the processor already has a page address stored as determined at decision operation 306, the processor performs write functions to memory in order to zero the page at the internally stored address. This is illustrated at operation 310.

Detailed Description Text (43):
In the embodiment of FIG. 5, a special command to "initialize page" or "zero page" is transmitted to the memory 208 via the address/command bus 510. Associated with the command are address and data signals that that can provide the address of the page that is to be zeroed. The memory can then execute a special function to clear the predetermined data segment (e.g., a cache line) starting at the address

provided. Alternatively, a separate hardware signal on line 552 can trigger the zeroing function at the address of the cache line or other data block to be zeroed.

CLAIMS:

9. The method of claim 1, further comprising initializing one or more of the inactive data blocks using the instructions of the instruction stream when an initialized inactive data block is required and no available computing cycles can be identified.

14. The method of claim 1, wherein identifying one or more inactive data blocks in the memory comprises identifying one or more inactive memory segments, wherein the memory segments comprise a predetermined block size of memory.

20. The method of claim 1, further comprising generating a second list of second addresses corresponding to the inactive data blocks which have been initialized and are available for use by the computing system.

WEST

Help | Logout | Interrupt

Main Menu | Search Form | Posting Counts | Show S Numbers | Edit S Numbers | Preferences | Cases

Search Results -

Term	Documents
(13 AND 14).USPT.	5
(L14 AND L13).USPT.	5

Database:
- US Patents Full-Text Database
- US Pre-Grant Publication Full-Text Database
- JPO Abstracts Database
- EPO Abstracts Database
- Derwent World Patents Index
- IBM Technical Disclosure Bulletins

Search: L15

Refine Search

Recall Text | Clear

Search History

DATE: Wednesday, November 12, 2003 Printable Copy Create Case

Set Name side by side	Query	Hit Count	Set Name result set
	DB=USPT; PLUR=YES; OP=ADJ		
L15	L14 and l13	5	L15
L14	identif$	626852	L14
L13	L12 and l11	5	L13
L12	task$1	155532	L12
L11	L9 and l3 and l2 and l1	7	L11
L10	L9 and l8 and l3 and l2 and l1	0	L10
L9	(memory or cache) and (start$ address and end$ address)	2691	L9
L8	task identif$	2066	L8
L7	L6 and l1 and l2 and l3	0	L7
L6	task entr$	176	L6
L5	L4 and l3 and l2 and l1	0	L5
L4	task$1 near5 (tabl$1 near3 entr$)	187	L4
L3	start$ address$2 and end$ address$2	2776	L3
L2	block size	7848	L2
L1	external cache	1188	L1

END OF SEARCH HISTORY

WEST

Help | Logout | Interrupt

Main Menu | Search Form | Posting Counts | Show S Numbers | Edit S Numbers | Preferences | Cases

Search Results -

Term	Documents
(29 AND 20).USPT.	8
(L29 AND L20).USPT.	8

Database: US Patents Full-Text Database
US Pre-Grant Publication Full-Text Database
JPO Abstracts Database
EPO Abstracts Database
Derwent World Patents Index
IBM Technical Disclosure Bulletins

Search: L30

Refine Search

Recall Text | Clear

Search History

DATE: Wednesday, November 12, 2003 Printable Copy Create Case

Set Name side by side	Query	Hit Count	Set Name result set
DB=USPT; PLUR=YES; OP=ADJ			
L30	L29 and l20	8	L30
L29	block same (start$ address and end$ address)	665	L29
L28	multi process$	5937	L28
L27	L26 and l22	21	L27
L26	task near3 identif$	3747	L26
L25	task identif$	2066	L25
L24	L23 and l22	22	L24
L23	block size$1	8268	L23
L22	l21 and l4	125	L22
L21	L20 and l19 and l12 and l9	133	L21
L20	task near4 manag$	5205	L20
L19	l10 near3 address	6728	L19
L18	l2 and l13	0	L18
L17	l2 and l12	1	L17
L16	l2 and l11	1	L16
L15	l2 and l10	1	L15
L14	l2 and l9	1	L14
L13	task near2 status$1	933	L13
L12	block$1	969300	L12
L11	start$	963760	L11
L10	cache	27558	L10
L9	identif$	626852	L9
L8	L7 and l2	0	L8
L7	identif$ and start address and block size and cache	192	L7
L6	L5 and l2	1	L6
L5	L4 and l1	78740	L5
L4	table or entr$	1120605	L4
L3	L2 and l1	1	L3
L2	5404483.pn.	1	L2
L1	task$1	155532	L1

END OF SEARCH HISTORY

WEST

Help | Logout | Interrupt

Main Menu | Search Form | Posting Counts | Show S Numbers | Edit S Numbers | Preferences | Cases

Search Results -

Term	Documents
(1 AND 62).USPT.	1
(L62 AND L1).USPT.	1

Database: US Patents Full-Text Database; US Pre-Grant Publication Full-Text Database; JPO Abstracts Database; EPO Abstracts Database; Derwent World Patents Index; IBM Technical Disclosure Bulletins

Search: L63

Refine Search | Recall Text | Clear

Search History

DATE: Monday, November 17, 2003 Printable Copy Create Case

Set Name side by side	Query	Hit Count	Set Name result set
DB=USPT; PLUR=YES; OP=ADJ			
L63	L62 and l1	1	L63
L62	internal or external	1117553	L62
L61	L60 and register	1	L61
L60	block size and task and l1	1	L60
L59	L58 and (l1 or l18 or l22)	3	L59
L58	bus near5 interface	39218	L58
L57	L53 and l22	0	L57
L56	L53 and l18	0	L56
L55	L53 andl18	0	L55
L54	L53 andl1	0	L54
L53	shar$	452159	L53

L52	L51 and l11	1	L52
L51	update$	125637	L51
L50	l22 and l43	0	L50
L49	l18 and l43	0	L49
L48	l1 and l22	0	L48
L47	l1 and l18	0	L47
L46	l1 and l43	0	L46
L45	L44 and l1	0	L45
L44	L43 near5 block$1	3	L44
L43	in use	2731	L43
L42	L40 and l18	1	L42
L41	L40 and l22	1	L41
L40	start address	7584	L40
L39	L38 and l1	0	L39
L38	task entr$	176	L38
L37	L33 and l18	0	L37
L36	L33 and l22	0	L36
L35	L33 and l1	1	L35
L34	L33 andl1	0	L34
L33	busy	26844	L33
L32	L29 and l22	0	L32
L31	L29 and l18	0	L31
L30	L29 and l1	0	L30
L29	busy near10 block$1	1627	L29
L28	busy near 10 block$1	0	L28
L27	l21 and (l1 or l18 or l22)	0	L27
L26	L1 and l21	0	L26
L25	flag	81500	L25
L24	L18 and l21	0	L24
L23	L22 and l21	0	L23
L22	5138696.pn.	1	L22
L21	flag near5 busy	994	L21
L20	L1 and l17	0	L20
L19	L18 and l17	0	L19
L18	6173385.pn.	1	L18
L17	free block$1	785	L17
L16	command	194632	L16
L15	L14 and l1	0	L15
L14	task command	406	L14
L13	L12 and address	1	L13

L12	task and l11	1	L12
L11	L10 and l1	1	L11
L10	RAM	166129	L10
L9	L8 and l1	0	L9
L8	cache	27558	L8
L7	L6 and l1	0	L7
L6	cache and address	22409	L6
L5	L4 and l1	1	L5
L4	block size	7848	L4
L3	L2 and l1	1	L3
L2	task ID	252	L2
L1	6128307.pn.	1	L1

END OF SEARCH HISTORY

WEST

Help | Logout | Interrupt

Main Menu | Search Form | Posting Counts | Show S Numbers | Edit S Numbers | Preferences | Cases

Search Results -

Term	Documents
(35 AND 38).USPT.	1
(L38 AND L35).USPT.	1

Database: US Patents Full-Text Database; US Pre-Grant Publication Full-Text Database; JPO Abstracts Database; EPO Abstracts Database; Derwent World Patents Index; IBM Technical Disclosure Bulletins

Search: L39

Refine Search | Recall Text | Clear

Search History

DATE: Sunday, June 01, 2003 Printable Copy Create Case

Set Name side by side	Query	Hit Count	Set Name result set
DB=USPT; PLUR=YES; OP=ADJ			
L39	L38 and l35	1	L39
L38	task$1 near3 list$1	1501	L38
L37	L36 and l35	0	L37
L36	task$1 near3 table	1243	L36
L35	L33 and l29	26	L35
L34	L33 and l30	1	L34
L33	cache coherency	1544	L33
L32	chache coherency	0	L32
L31	L30 and l29	1	L31
L30	task cycle$1	50	L30

L29	L28 and l8	57	L29
L28	L27 and l12 and l13	203	L28
L27	L26 and l18	524	L27
L26	l6 and l24	813	L26
L25	L24 and l16	1	L25
L24	memory near3 allocat$	13086	L24
L23	command$1 and l16	1	L23
L22	l12 and l13 and l16	1	L22
L21	L18 and l15	1	L21
L20	L18 and l17	1	L20
L19	L18 and l16	1	L19
L18	star$ and end$ and size and block$1	174226	L18
L17	L16 and l1	1	L17
L16	5404483.pn.	1	L16
L15	5404482.pn.	1	L15
L14	L13 and l12 and l11	13	L14
L13	exclusive	84666	L13
L12	shared	76373	L12
L11	L10 and l8 and l6 and l4	23	L11
L10	block near3 allocat$	4988	L10
L9	L8 and l6.ab.	6	L9
L8	multi process$	5573	L8
L7	L6 and l5 and l4 and l2	7	L7
L6	task$1 near3 manag$	4227	L6
L5	cache address	1433	L5
L4	identifier$1	41450	L4
L3	task entr$	174	L3
L2	block$1 near2 (size or starting address or ending address)	16761	L2
L1	block$1 and (size or starting address or ending address)	442466	L1

END OF SEARCH HISTORY

WEST

Generate Collection Print

L14: Entry 8 of 13 File: USPT Apr 4, 1995

DOCUMENT-IDENTIFIER: US 5404482 A
TITLE: Processor and method for preventing access to a locked memory block by recording a lock in a content addressable memory with outstanding cache fills

Parent Case Text (2):
The present application is a continuation-in-part of Ser. No. 07/547,699, filed Jun. 29, 1990, entitled BUS PROTOCOL FOR HIGH-PERFORMANCE PROCESSOR, by Rebecca L. Stamm et al., now abandoned in favor of continuation application Ser. No. 08/034,581, filed Mar. 22, 1993, entitled PROCESSOR SYSTEM WITH WRITEBACK CACHE USING WRITEBACK AND NON WRITEBACK TRANSACTIONS STORED IN SEPARATE QUEUES, by Rebecca L. Stamm, et al., issued on May 31, 1994, as U.S. Pat. No. 5,317,720, and Ser. No. 07/547,597, filed Jun. 29, 1990, entitled ERROR TRANSITION MODE FOR MULTI-PROCESSOR SYSTEM, by Rebecca L. Stamm et al., issued on Oct. 13, 1992, as U.S. Pat. No. 5,155,843, incorporated herein by reference.

Brief Summary Text (3):
This invention is directed to digital computers, and more particularly to a multi-processor system in which the processors share a common memory, but in which a processor may obtain an exclusive right to access a respective block of memory. The invention specifically relates to such a multi-processor system in which each processor has a cache memory and follows a cache coherency protocol.

Brief Summary Text (5):
Processors in a multi-processor computer system typically communicate via a shared memory. To improve system performance, each processor has a cache memory for temporarily storing copies of data being accessed. Such a hierarchical memory system may follow either a "write through" or a "write back" protocol. In a "write through" protocol, a processor immediately writes data to the shared memory so that any other processor may fetch the most recent memory state from the shared memory. In a "writeback" protocol, a processor writes data to its cache, but this new memory state is written back to the shared memory only when the memory space in the cache needs to be used for different addresses in a cache fill operation, or when another processor needs the new memory state. Therefore, the writeback protocol reduces the number of memory access operations to the shared memory when the new memory state is not needed by the other processors. In general, the write through protocol is preferred when the processors frequently access the same memory addresses, and the write back protocol is preferred when the processors infrequently access the same memory addresses.

Brief Summary Text (6):
Whenever processors communicate via a shared memory, it is desirable to require the processors to follow a protocol insuring that a memory address is not written to simultaneously by more than one processor, or else the result of one processor will be nullified by the result of another processor. Such synchronization of memory access is commonly achieved by requiring a processor to obtain an exclusive privilege to write to an addressed portion of the shared memory, before executing a write operation. In a multi-processor system employing writeback caches, such an exclusive privilege gives rise to a cache coherency problem in which data written in the cache of a processor having such an exclusive privilege might be the only valid copy of data for the addressed portion of memory. A cache coherency protocol is required which permits a processor to obtain readily the valid copy of data as well as the privilege to write to it.

Brief Summary Text (7):
One known cache coherency protocol for a multi-processor system employing writeback caches is based on the concept of block ownership; an addressed portion of memory the size of a cache block is either owned by the shared memory or it is owned by one of the writeback caches. Only one of the processors, or the shared memory, may own the block of memory at any given time, and this ownership is indicated by an ownership bit for each block in the shared memory and in each of the caches. A processor may write to a block only when the processor owns the block. Therefore the ownership bits always identify a unique "valid" block in the system. Shared read-only access to a block is permitted only when the shared memory owns the block. To indicate whether a processor may read a block, each of the caches includes, for each block, a "valid" bit. When a processor desires to read a block that is not valid in its cache, it issues a read transaction to the shared memory, requesting the shared memory to fill its cache with valid data. When a processor desires to write to a block which it does not own, it issues an ownership-read transaction to the shared memory, requesting ownership as well as a fill. From the perspective of the other processors, these transactions are cache coherency transactions, which request any other processor having ownership to give up ownership and writeback the data of the requested block, and in the case of an ownership read transaction, further request the other processors to invalidate any copies of the requested block.

Brief Summary Text (8):
The architecture of the instruction set for a digital computer typically includes certain "interlocked" instructions that are guaranteed to perform atomic operations upon memory in a multi-processing environment. Such "interlocked" instructions, for example, include operands having an access type of "modify", wherein an operand is first read from memory as a source operand, modified, and written back to memory as a destination operand. In a multi-processor system, the access type of "modify" raises the possibility that another CPU might access memory between the time that an operand is read from memory and the time that the operand is modified and written back to memory, leading to a result in memory which might not appear consistent under certain program sequences. To prevent this problem, interlocked instructions have been executed in a multi-processor environment by using the execution unit to request fetching of the operand and to request a memory "read lock" when fetching the second operand from memory, and to request a memory "write unlock" when putting the result back to memory.

Brief Summary Text (9):
Typically, the time for a cache coherency transaction to be transmitted over a system bus is much shorter than the time for fill data to be retrieved from the shared memory. Therefore system performance can be improved by use of a pended bus (i.e., a bus which permits more than one transaction to be pending on the bus at any given time). The use of such a "pended" bus, however, raises the possibility of receiving a number of cache coherency transactions when a cache fill is outstanding. To obtain the improvement in performance of the pended bus, the cache coherency transactions should be executed as soon as possible, but the cache coherency transactions may conflict with the outstanding fill as well as an outstanding lock.

Drawing Description Text (3):
FIG. 1 is a block diagram of a multi-processor computer system incorporating the present invention;

Detailed Description Text (2):
The Multi-Processor System

Detailed Description Text (3):
Referring to FIG. 1, according to one embodiment, a multi-processor computer system employing features of the invention includes a central processing unit (CPU) chip or module 10 connected by a system bus 11 to a system memory 12, an input/output (I/O) unit 13c, and to additional CPU's 28. As will be further described below with reference to FIG. 6, two I/O units 13a, 13b may also be connected directly to a bus 20. In a preferred embodiment, the CPU 10 is formed on a single integrated circuit, although the present invention may be used with a CPU implemented as a chip set

mounted on a single circuit board or multiple boards.

Detailed Description Text (17):
The present invention more particularly concerns the operation of the cache-controller 26 and maintenance of coherency of the back-up cache 15 with the memory 12 and caches of the other CPU's 28 in the multi-processor system in FIG. 1. Therefore, the specific construction of the components in the CPU 10 other than the cache controller 26 are not pertinent to the present invention. The reader, however, may find additional details in the above-referenced U.S. application Ser. No. 07/547,597, filed Jun. 29, 1990, and issued on Oct. 13, 1992, as U.S. Pat. No. 5,155,843, incorporated herein by reference.

Detailed Description Text (19):
Cache coherency in the multi-processor system of FIG. 1 is based upon the concept of ownership; a hexaword (16-word) block of memory may be owned either by the system memory 12 or by a backup cache 15 in a CPU on the bus 11--in a multiprocessor system. Only one of the caches, or system memory 12, may own the hexaword block at a given time, and this ownership is indicated by an ownership bit for each hexaword in both memory 12 and the backup cache 15 (1 for own, 0 for not-own).

Detailed Description Text (20):
Shared read-only access to a block among the CPUs 10, 28 is permitted only when system memory 12 owns the block. A CPU may write to a block only when the CPU owns the block. These rules ensure that there is always a unique "valid" block of data in the system, identified by the ownership bits in the caches, and a CPU will always read data from the valid block and write data to the valid block.

Detailed Description Text (21):
The multi-processor system follows certain protocols which ensure rapid access to the valid data of an addressed block. Each back-up cache 15 maintains two bits associated with each cache block. These two bits are called VALID and OWNED, and they determine the state of the cache block as shown in TABLE A.

Detailed Description Text (30):
The preferred embodiment of FIG. 1, however, does have one instance where one CPU will not immediately relinquish ownership to another CPU. The preferred embodiment executes VAX instructions, including certain "interlocked" instructions that are guaranteed to perform atomic operations upon memory in a multi-processing environment. An example is an "add aligned word interlocked" instruction (ADAWI) which adds a first operand to a second operand and sets the second operand to the sum. The destination operand has an access type of "modify" raising the possibility that one CPU might obtain ownership of a cache block between the time that the second operand is read from memory and the time that the second operand is modified and written back to memory, leading to a result in memory which might not appear consistent under certain program sequences. Computers which execute (VAX) instructions in a multi-processing environment typically prevent such an interruption of memory access by using the execution unit to request fetching of the second operand and to request a memory "read lock" when fetching the second operand from memory, and to request a memory "write unlock" when putting the result back to memory.

Detailed Description Text (57):
Referring to FIG. 6, the bus 20 consists of a number of lines in addition to the 64-bit, multiplexed address/data lines 20a which carry the addresses and data in alternate cycles as seen in trace (a) of FIG. 5. The lines shared by the nodes on the bus 20 (the CPU 10, the I/O 13a, the I/O 13b and the interface chip 21) include the address/data bus 20a, a four-bit command bus 20b which specifies the current bus transaction during a given cycle (write, instruction stream read, data stream read, etc.), a three-bit ID bus 20c which contains the identification of the bus commander during the address and return data cycles (each commander can have two read transactions outstanding), a three-bit parity bus 20d, and the acknowledge line 20e. All of the command encodings for the command bus 20b and definitions of these transactions are set forth in Table A, below. The CPU also supplies four-phase bus clocks from the clock generator 30 on lines 20f.

Detailed Description Text (58):
In addition to these shared lines in the bus 20, each of the three active nodes CPU 10, I/O 13a and I/O 13b individually has the request, hold and grant lines 20g, 20h and 20i, connecting to the arbiter 325 in the memory interface chip 21. A further function is provided by a suppress line 20j, which is asserted by the CPU 10, for example, in order to suppress new transactions on the bus 20 that the CPU 10 treats as cache coherency transactions. It does this when its two-entry cache coherency queue 61 is in danger of overflowing. During the cycle when the CPU 10 asserts the suppress line 20j, the CPU 10 will accept a new transaction, but transactions beginning with the following cycle are suppressed (no node will be granted command of the bus). While the suppress line 20j is asserted, only fills and writebacks are allowed to proceed from any nodes other than the CPU 10. The CPU 10 may continue to put all transactions onto the bus 20 (as long as WB-only line 20k is not asserted). Because the in-queue 61 is full and takes the highest priority within the cache controller unit 26, the CPU 10 is mostly working on cache coherency transactions while the suppress line 20j is asserted, which may cause the CPU 10 to issue write-disowns on the bus 20. However, the CPU 10 may and does issue any type of transaction while its suppress line 20j is asserted. The I/O nodes 13a and 13b have a similar suppress line function.

Detailed Description Text (68):
As described above, the bus 20 is a 64-bit, pended, multiplexed address/data bus, synchronous to the CPU 10, with centralized arbitration provided by the interface chip 21. Several transactions may be in process at a given time, since a Read will take several cycles to produce the read-return data from the memory 12 and meanwhile other transactions may be interposed. Arbitration and data transfer occur simultaneously (in parallel) on the bus 20. Four nodes are supported: the CPU 10, the system memory (via bus 11 and interface chip 21) and two I/O nodes 13a and 13b. On the 64-bit bus 20a, data cycles (64-bits of data) alternate with address cycles containing 32-bit addresses plus byte masks and data length fields; a parallel command and arbitration bus carries a command on lines 20b, an identifier field on lines 20c defining which node is sending, and an Ack on line 20e; separate request, hold, grant, suppress and writeback-only lines are provided to connect each node to the arbiter 325.

Detailed Description Text (70):
The backup cache 15 for the CPU 10 is a "write-back" cache, so there are times when the backup cache 15 contains the only valid copy of a certain block of data, in the entire multi-processor system of FIG. 1. The backup cache 15 (both tag store and data store) is protected by ECC. Check bits are stored when data is written to the cache 15 data RAM or written to the tag RAM, then these bits are checked against the data when the cache 15 is read, using ECC check circuits 330 and 331 of FIG. 4. When an error is detected by these ECC check circuits, an Error Transition Mode is entered by the C-box controller 306; the backup cache 15 can't be merely invalidated, since other system nodes 28 may need data owned by the backup cache 15. In this error transition mode, the data is preserved in the backup cache 15 as much as possible for diagnostics, but operation continues; the object is to move the data for which this backup cache 15 has the only copy in the system, back out to memory 12, as quickly as possible, but yet without unnecessarily degrading performance. For blocks (hexawords) not owned by the backup cache 15, references from the memory management unit 25 received by the cache controller unit 26 are sent to memory 12 instead of being executed in the backup cache 15, even if there is a cache hit. For blocks owned by the backup cache 15, a write operation by the CPU 10 which hits in the backup cache 15 causes the block to be written back from backup cache 15 to memory 12, and the write operation is also forwarded to memory 12 rather than writing to the backup cache 15; only the ownership bits are changed in the backup cache 15 for this block. A read hit to a valid-owned block is executed by the backup cache 15. No cache fill operations are started after the error transition mode is entered. Cache coherency transactions from the system bus 20 are executed normally, but this does not change the data or tags in the backup cache 15, merely the valid and owned bits. In this manner, the system continues operation, yet the data in the backup cache 15 is preserved as best it can be, for later diagnostics.

Detailed Description Text (184):
Preferably, the data RAM control is a state machine which executes any of the

following tasks, upon instruction from the arbiter: DAT.sub.-- DREAD (reads four quadwords of data-stream data from the back-up cache 15 and sends them to the memory management unit); DAT.sub.-- IREAD (reads four quadwords of instruction-stream data from the back-up cache 15 and sends them to the memory management unit, and the task may be cancelled midstream if the IREAD is aborted by the memory management unit); DAT.sub.-- WB (reads four quadwords of data from the back-up cache 15 and sends them to the write-back queue (63 in FIG. 4); DAT.sub.-- RM.sub.-- WRITE (performs a read-modify-write operation on a quadword in the back-up cache 15); DAT.sub.-- WRITE.sub.-- BMO (performs a full quadword write on the back-up cache); and DAT.sub.-- FILL (writes fill data into the back-up cache 15, and merges write data with the fill, if necessary). When the data RAM control 474 has finished executing a task, the data RAM control notifies the arbiter 471.

Detailed Description Text (186):
In a first step 481, the arbiter gives highest priority to performing a de-allocate caused by a previous task. When a transaction such as a read miss causes a cache block to be de-allocated, this de-allocate always takes place in step 482 as the next data RAM task. In step 483, transactions in the in queue 61 are given the next-highest priority. Fills and cache coherency requests both arrive in the in queue 61, and then in step 484, the fill or cache coherency transaction at the head of the in queue is performed.

Detailed Description Text (191):
Turning now to FIG. 19, there is shown a flow chart of the steps followed by the arbiter 471 in servicing the D-read latch 299, the I-read latch 300, and the write queue 60. In steps 501, 502, 503, the source given priority asserts the address of its memory command upon the internal address bus 288 of the cache controller (see FIG. 4). If the memory command accesses an internal processor register (IPR) or I/O or a write unlock, as tested in step 504, then the command is completed in step 505. (To simplify implementation, the test in step 504 can be done concurrently with step 506 so that the fill CAM is always addressed and a hit always causes execution of a command other than a WRITE UNLOCK to stall.) If, however, the memory command accesses memory space, then in step 506, processing of the task is halted if there is a hit in the fill CAM. In this case, the memory space access conflicts with an outstanding fill or an outstanding READ LOCK. If, however, there is not a conflict with an outstanding fill or READ LOCK, then in step 507, execution branches depending on whether the cache is in the above-described error transition mode or whether the memory access is requested by an ownership command. If so, then in step 508, the tag RAMs are accessed to determine whether there is a cache hit in an owned block. If not, then if the cache is in the error transition mode, as tested in step 509, the back-up cache is bypassed and the read or write is sent directly to system memory (12 in FIG. 1) in step 510. If, however, in step 509, the cache was not operating in the error transition mode, then in step 511, an ownership read is sent to memory, and in step 512, the fill CAM is set to record that the refill is in progress. Moreover, if the addressed block in the cache is owned, as tested in step 513, then in step 514, the cache block is de-allocated and written back to memory in the next task. In other words, in step 514, a flag is set which is inspected by the arbitrator in step 481 of FIG. 18 to determine whether a need to de-allocate was caused by the previous task.

Detailed Description Text (192):
If in step 508 there was a cache hit in an owned block of the back-up cache, then in step 515, execution branches depending on whether the cache is in the error transition mode. If so, then in step 516, an ownership transaction is sent to memory, and the memory block is de-allocated and written back to memory in the next task. From step 515 or 516, execution continues in step 517 to complete the command.

Detailed Description Text (193):
If in step 507 it was found that the cache was neither in the error transition mode nor the command was an ownership command, then in step 518, execution branches depending on whether there was a cache hit. If so, then the command is completed in step 517. If not, then execution branches to step 519, where a refill of the cache block is begun by sending a data read or instruction read to memory. The fact that the refill is in progress is recorded in the fill CAM in step 512, and if the

address block in the cache is owned, as tested in step 513, then in step 514, the address block is de-allocated and written back to memory in the next task.

Other Reference Publication (1):
Archibald et al., "Cache Coherence Protocols: Evaluation Using a Multi-Processor Simulation Model," ACM Transactions on Computer Systems, No. 4, Nov. 1986, New York, N.Y., U.S.A., pp. 273-298.

CLAIMS:

15. A processor for a multi-processor computer system, said multi-processor computer system having system bus for coupling processors to a system memory, said system bus operating in accordance with a block ownership cache coherency protocol, said processor comprising, in combination:

instruction decoding means for decoding computer program instructions to generate requests for reading data at specified read addresses;

instruction execution means connected to said instruction decoding means for executing the computer program instructions decoded by said instruction decoding means to generate requests for writing data at specified write addresses;

means, coupled to said instruction decoding means and to said instruction execution means, for generating a memory lock request for a memory lock upon a specified memory lock address, and for generating a corresponding memory unlock request;

a cache memory for storing blocks of data, and in association with each block of data, a memory address, an indication of whether each block is valid for providing data from said memory address in response to said requests for reading data, and an indication of whether each block is valid for receiving data from said requests for writing data to said memory address;

a content addressable memory including a plurality of entries and means for storing in each entry said memory lock address together with an indication of a read lock in progress or a fill address of a fill request to a shared memory in said multi-processor system requesting fill data from said fill address in said shared memory, an indication of whether the fill request is in progress, an indication of whether the fill address is associated with a request for validation for writing data to said fill address, an indication of whether a read invalidate request was received, before return of said fill data, from another processor in said multi-processor system requesting invalidation of any indication that a cache block having said fill address in said cache memory is valid for receiving write data, and an indication of whether an ownership-read invalidate request was received, before return of said fill data, from another processor in said multi-processor system requesting invalidation of any indication that a cache block having said fill address is valid for providing read data,

means, responsive to a request for reading data from a specified read address, for addressing said cache memory with said read address, for reading data from said cache memory when said cache memory contains a cache block having said read address and indicated as valid for providing read data, and when said cache memory does not contain a cache block having said read address and indicated as valid for providing read data, for sending a fill request to said system memory including said read address and for storing said read address in said content addressable memory,

means, responsive to a request for writing data to a specified write address, for writing data to said cache memory when said cache memory contains a cache block having said write address and indicated as valid for receiving write data, and when said cache memory does not contain a cache block having said write address and indicated as valid for receiving write data, for sending a fill request to said system memory including said write address and a request for validation for a write operation, and for storing in said content addressable memory said write address together with an indication that the fill address is associated with a request for validate for a write operation;

means, responsive to receiving from another processor in said multi-processor system a read invalidate request having a specified read invalidate address, for addressing said content addressable memory with said specified read invalid data address, and when a fill address matching said specified read invalid address is found in said content addressable memory, for setting said indication of whether a read invalidate request was received from another processor in said multi-processor system before release of said memory lock or return of said fill data;

means, responsive to receiving from another processor in said multi-processor system an ownership-read invalidate request having a specified ownership-read invalidate address, for addressing said content addressable memory with said specified ownership-read invalidate address, and when a fill address matching said specified ownership-read invalidating address is found in said content addressable memory, for setting said indication of whether an ownership-read invalidate request was received from another processor in said multi-processor system before release of said memory lock or return of said fill data;

first means, responsive to return of said fill data or release of said memory lock, for checking said indication in a corresponding entry of said content addressable memory of whether an ownership-read invalidate request was received during said memory lock or before return of said fill data, and when an ownership-read invalidate request was received during said memory lock or before return of said fill data, for invalidating an indication that a cache block having the fill address in said cache memory is valid for providing read data;

second means, responsive to return of said fill data or release of said memory lock, for checking said indication in a corresponding entry of said content addressable memory of whether a read invalidate request was received during said memory lock or before return of said fill data, and when a read invalidate request was received during said memory lock or before return of said fill data, for invalidating an indication that a cache block having the fill address in said cache memory is valid for receiving write data.

WEST

End of Result Set

Generate Collection | Print

L22: Entry 1 of 1 File: USPT Apr 4, 1995

DOCUMENT-IDENTIFIER: US 5404483 A
TITLE: Processor and method for delaying the processing of cache coherency transactions during outstanding cache fills

Abstract Text (1):
A processor and method for delaying the processing of cache coherency transactions during outstanding cache fills in a multi-processor system using a shared memory. A first processor fetches data having a specified address by addressing a cache memory, and when the specified address is not in the cache, saving the specified address in a fill address memory, and sending a fill request to the shared memory. Before return of fill data, the first processor receives a cache coherency request including the specified address from a second processor requesting invalidation of an addressed block of data. The first processor responds by checking whether the fill address memory includes the specified address, and upon finding the specified address in the fill address memory, delaying execution of the cache coherency request until the fill data is returned, and when the fill data is returned, using the fill data without retaining a validated block of the fill data in the cache. In a preferred embodiment, the fill memory is a content-addressable memory including a plurality of entries, and each entry has a fill address, an ownership fill bit (OREAD), an ownership-read invalidate pending bit (OIP), and a read invalidate pending bit (RIP). The OIP or RIP bit is set when execution of a cache coherency request is delayed, and these bits are read upon completion of a fill to execute the delayed request.

US Patent No. (1):
5404483

Brief Summary Text (6):
Processors in a multi-processor computer system typically communicate via a shared memory. To improve system performance, each processor has a cache memory for temporarily storing copies of data being accessed. Such a hierarchical memory system may follow either a "write through" or a "write back" protocol. In a "write through" protocol, a processor immediately writes data to the shared memory so that any other processor may fetch the most recent memory state from the shared memory. In a "writeback" protocol, a processor writes data to its cache, but this new memory state is written back to the shared memory only when the memory space in the cache needs to be used for different addresses in a cache fill operation, or when another processor needs the new memory state. Therefore the writeback protocol reduces the number of memory access operations to the shared memory when the new memory state is not needed by the other processors. In general, the write through protocol is preferred when the different processors frequently access the same shared memory addresses, and the write back protocol is preferred when the different processors infrequently access the same shared memory addresses.

Brief Summary Text (7):
Whenever processors communicate via a shared memory, it is desirable to require the processors to follow a protocol ensuring that a memory address is not written to simultaneously by more than one processor, or else the result of one processor will be nullified by the result of another processor. Such synchronization of memory access is commonly achieved by requiring a processor to obtain an exclusive privilege to write to an addressed portion of the shared memory, before executing a

write operation. In a multi-processor system employing writeback caches, such an exclusive privilege gives rise to a cache coherency problem in which data written in the cache of a processor having such an exclusive privilege might be the only valid copy of data for the addressed portion of memory. A cache coherency protocol is required which permits a processor to obtain readily the valid copy of data as well as the privilege to write to it.

Brief Summary Text (8):
One known cache coherency protocol for a multi-processor system employing writeback caches is based on the concept of block ownership; an addressed portion of memory the size of a cache block is either owned by the shared memory or it is owned by one of the writeback caches. Only one of the processors, or the shared memory, may own the block of memory at any given time, and this ownership is indicated by an ownership bit for each block in the shared memory and in each of the caches. A processor may write to a block only when the processor owns the block. Therefore the ownership bits always identify a unique "valid" block in the system. Shared read-only access to a block is permitted only when the shared memory owns the block. To indicate whether a processor may read a block, each of the caches includes, for each block, a "valid" bit. When a processor desires to read a block that is not valid in its cache, it issues a read transaction to the shared memory, requesting the shared memory to fill its cache with valid data. When a processor desires to write to a block which it does not own, it issues an ownership-read transaction to the shared memory, requesting ownership as well as a fill. From the perspective of the other processors, these transactions are cache coherency transactions, which request any other processor having ownership to give up ownership and writeback the data of the requested block, and in the case of an ownership read transaction, further request the other processors to invalidate any copies of the requested block.

Brief Summary Text (9):
Typically the time for a cache coherency transaction to be transmitted over a system bus is much shorter than the time for fill data to be retrieved from the shared memory. Therefore system performance can be improved by use of a pended bus (i.e., a bus which permits more than one transaction to be pending on the bus at any given time). The use of such a "pended" bus, however, leads to a problem of data coherency where a processor may issue a read transaction to fill a cache block, but before receiving the fill data, the processor may receive a cache coherency transaction requesting the cache block to be disowned or invalidated. A conventional way of handling this problem is for the processor to immediately invalidate the cache block by clearing the "valid" bit, which effectively discards the fill data when it is received. But in this case the processor which first requested the data does not get to use it, and must reissue a read transaction to get the data.

Brief Summary Text (11):
In accordance with a basic aspect of the invention, there is provided a method of operating a first processor in a multi-processor system that has a second processor and a shared memory accessed by both of the first and second processors. The first processor has a cache memory for storing blocks of data and associated addresses. The first processor fetches data having a specified address by addressing the cache memory with the specified address, and when the specified address is not found in the cache memory, saving the specified address in a fill address memory, and sending a fill data request including the specified address to the shared memory. Before receipt of the requested fill data from the shared memory, the first processor receives a cache coherency request including the specified address from the second processor requesting invalidation of a block of data having the specified address. The first processor responds to the cache coherency request by checking whether the fill memory includes the specified address, and upon finding that the fill memory includes the specified address, delaying execution of the cache coherency request until the fill data is received by the first processor, and when the fill data is received by the first processor, using the fill data without retaining a validated block of the fill data in the cache.

Brief Summary Text (12):
In a preferred embodiment, the fill memory is a content-addressable memory (CAM) including a plurality of entries, and each of the entries has a fill address, and

associated with the fill address, an ownership fill bit (OREAD), an ownership-read invalidate pending bit (OIP), and a read invalidate pending bit (RIP). The ownership fill bit (OREAD) is set when the first processor requests a fill with ownership of a block of fill data having the associated fill address. The ownership-read invalidate pending (OIP) bit is set when the first processor receives from the second processor a request for ownership of the block of data having the associated fill address. The read invalidate pending (RIP) bit is set when the first processor receives from the second processor a request to read the block of data having the associated fill address, and the ownership fill bit (OREAD) associated with the fill address has been set. Upon receipt of a block of fill data from the shared memory, the first processor loads the block of fill data into the cache memory and subsequently uses the fill data for a data processing operation. Immediately after loading the block of fill data into the cache memory, however, the block of fill data is invalidated when the associated OIP bit has been set, and the block of fill data is disowned and written back to the shared memory when the associated OREAD bit has been set and the associated OIP or RIP bit has been set. Finally, the first processor clears the entry in the fill memory having the address of the block of fill data.

Detailed Description Text (20):
Shared read-only access to a block among the CPUs 10, 28 is permitted only when system memory 12 owns the block. A CPU may write to a block only when the CPU owns the block. These rules ensure that there is always a unique "valid" block of data in the system, identified by the ownership bits in the caches, and a CPU will always read data from the valid block and write data to the valid block.

Detailed Description Text (57):
Referring to FIG. 6, the bus 20 consists of a number of lines in addition to the 64-bit, multiplexed address/data lines 20a which carry the addresses and data in alternate cycles as seen in trace (a) of FIG. 5. The lines shared by the nodes on the bus 20 (the CPU 10, the I/O 13a, the I/O 13b and the interface chip 21) include the address/data bus 20a, a four-bit command bus 20b which specifies the current bus transaction during a given cycle (write, instruction stream read, data stream read, etc.), a three-bit ID bus 20c which contains the identification of the bus commander during the address and return data cycles (each commander can have two read transactions outstanding), a three-bit parity bus 20d, and the acknowledge line 20e. All of the command encodings for the command bus 20b and definitions of these transactions are set forth in Table A, below. The CPU also supplies four-phase bus clocks from the clock generator 30 on lines 20f.

Detailed Description Text (58):
In addition to these shared lines in the bus 20, each of the three active nodes CPU 10, I/O 13a and I/O 31b individually has the request, hold and grant lines 20g, 20h and 20i, connecting to the arbiter 325 in the memory interface chip 21. A further function is provided by a suppress line 20j, which is asserted by the CPU 10, for example, in order to suppress new transactions on the bus 20 that the CPU 10 treats as cache coherency transactions. It does this when its two-entry cache coherency queue 61 is in danger of overflowing. During the cycle when the CPU 10 asserts the suppress line 20j, the CPU 10 will accept a new transaction, but transactions beginning with the following cycle are suppressed (no node will be granted command of the bus). While the suppress line 20j is asserted, only fills and writebacks are allowed to proceed from any nodes other than the CPU 10. The CPU 10 may continue to put all transactions onto the bus 20 (as long as WB-only line 20k is not asserted). Because the in-queue 61 is full and takes the highest priority within the cache controller unit 26, the CPU 10 is mostly working on cache coherency transactions while the suppress line 20j is asserted, which may cause the CPU 10 to issue write-disowns on the bus 20. However, the CPU 10 may and does issue any type of transaction while its suppress line 20j is asserted. The I/O nodes 13a and 13b have a similar suppress line function.

CLAIMS:

3. The method as claimed in claim 1, wherein said cache coherency request is a read invalidate request requesting said first processor to relinquish any exclusive privilege to write to said block of data having said specified memory address, and wherein said method includes clearing an indication of ownership associated with

said block of said fill data in said cache memory so that a block of said fill data validated for writing is not retained in said cache memory, and writing said block of said fill data in said cache memory back to said system memory.

8. The method as claimed in claim 7, wherein said data processing operation includes the writing of data to said specified memory address, said fill data request includes a request for an exclusive privilege to write to said block of data having said specified memory address, and wherein said method further includes writing said block of said fill data in said cache memory back to said system memory.

12. The method as claimed in claim 10, wherein said data processing operation includes the writing of data to said specified memory address, said fill data request includes a request for an exclusive privilege to write to said specified memory address, said method includes setting in said entry of said content addressable memory an indication of said request for an exclusive privilege to write to said specified memory address, said cache coherency request is a read invalidate request requesting said first processor to relinquish any exclusive privilege to write to a block of data having said specified memory address, said step (b) (ii) further includes checking whether said entry indicates said request for an exclusive privilege to write to said specified memory address, and wherein the setting in said content addressable memory of an indication that said cache coherency request is pending for said specified memory address is performed upon finding that said entry indicates said request for an exclusive privilege to write to said specified memory address.

13. The method as claimed in claim 10, wherein said data processing operation includes the writing of data to said specified memory address, said fill data request includes a request for an exclusive privilege to write to said specified memory address, said method includes setting in said entry of said content addressable memory an indication of said request for an exclusive privilege to write to said specified memory address, said cache coherency request is an ownership-read invalidate request requesting said first processor to refrain from reading or writing to a block of data having said specified memory address, and wherein step (c) further includes checking whether said entry indicates said request for an exclusive privilege to write to said specified memory address, and upon finding that said entry indicates said request for an exclusive privilege to write to said specified memory address, writing a block of fill data including data written in accordance with said data processing operation back to said system memory.

14. The method as claimed in claim 10, wherein said cache coherency request is a read invalidate request requesting said first processor to relinquish any exclusive privilege to write to a block of data having said specified memory address, said method includes storing said fill data in a cache block in said cache memory, and wherein the execution of said cache coherency request includes clearing an indication of ownership associated with said cache block in said cache memory storing said fill data, and writing said cache block of fill data back to said system memory.

16. A processor for a multi-processor computer system, said multi-processor computer system having a system bus for coupling processors to a system memory, said system bus operating in accordance with a block ownership cache coherency protocol, said processor comprising, in combination:

instruction decoding means for decoding computer program instructions to generate requests for reading data at specified read addresses;

instruction execution means connected to said instruction means for executing the computer program instructions decoded by said instruction decoding means to generate requests for writing data at specified write addresses;

a cache memory for storing blocks of data, and in association with each block of data, a memory address, an indication of whether each block is valid for providing data from said memory address in response to said requests for reading data, and an indication of whether each block is valid for receiving data from said requests for writing data to said memory address;

a content addressable memory including a plurality of entries and means for storing in each entry a fill address of a fill request to a system memory in said multi-processor system requesting fill data from said fill address in said shared memory, an indication of whether the fill address is associated with a request for validation for writing data to said fill address, an indication of whether a read invalidate request was received, before return of said fill data, from another processor in said multi-processor system requesting invalidation of any indication that a cache block having said fill address in said cache memory is valid for receiving write data of whether an ownership-read invalidate request was received, before return of said fill data, from another processor in said multi-processor system requesting invalidation of any indication that a cache block having said fill address is valid for providing read data;

means, responsive to a request for reading data from a specified read address, for addressing said cache memory with said read address, for reading data from said cache memory when said cache memory contains a cache block having said read address and indicated as valid for providing read data, and when said cache memory does not contain a cache block having said read address and indicated as valid for providing read data, for sending a fill request to said main memory including said read address and for storing said read address in said content addressable memory;

means, responsive to a request for writing data to a specified write address, for writing data to said cache memory when said cache memory contains a cache block having said write address and indicated as valid for receiving write data, and when said cache memory does not contain a cache block having said write address and indicated as valid for receiving write data, for sending a fill request to said system memory including said write address and a request for validation for a write operation, and for storing in said content addressable memory said write address together with an indication that the fill address is associated with a request for validation for a write operation;

means, responsive to receiving from another processor in said multi-processor system a read invalidate request having a specified read invalidate address, for addressing said content addressable memory with said specified read invalidate address, and when a fill address matching said specified read invalidate address is found in said content addressable memory, for setting the indication of whether a read invalidate request was received from another processor in said multi-processor system before return of said fill data;

means, responsive to receiving from another processor in said multi-processor system an ownership-read invalidate request having a specified ownership-read invalidate address, for addressing said content addressable memory with said specified ownership-read invalidate address, and when a fill address matching said specified ownership-read invalidating address is found in said content addressable memory, for setting the indication of whether an ownership-read invalidate request was received from another processor in said multi-processor system before return of said fill data;

first means, responsive to return of said fill data for checking said indication in said content addressable memory of whether an ownership-read invalidate request was received before return of said fill data, and when an ownership-read invalidate request was received before return of said fill data, for invalidating an indication that a cache block having the fill address in said cache memory is valid for providing read data; and second means, responsive to return of said fill data, for checking the indication in said content addressable memory of whether a read invalidate request was received before return of said fill data, and when a read invalidate request was received before return of said fill data, for invalidating an indication that a cache block having the fill address in said cache memory is valid for receiving write data.

WEST

End of Result Set

Generate Collection Print

L23: Entry 1 of 1 File: USPT Apr 4, 1995

DOCUMENT-IDENTIFIER: US 5404483 A
TITLE: Processor and method for delaying the processing of cache coherency transactions during outstanding cache fills

US Patent No. (1):
5404483

Drawing Description Text (21):
FIG. 19 is a flowchart showing control sequences followed by the arbiter of FIG. 17 when responding to memory commands from the memory management unit in FIG. 1; and

Detailed Description Text (6):
The CPU 10 accesses the backup cache 15 through a bus 19, separate from a CPU bus 20 used to access the system bus 11; thus, a cache controller 26 for the backup cache 15 is included within the CPU chip. Both the CPU bus 20 and the system bus 11 are 64-bit bidirectional multiplexed address/data buses, accompanied by control buses containing request, grant, command lines, etc. The bus 19, however, has a 64-bit data bus and separate address buses. The system bus 11 is interconnected with the CPU bus 20 by an interface/arbiter unit 21 (hereinafter referred to as interface 21) functioning to arbitrate access by the CPU 10 and the other components on the CPU bus 20.

Detailed Description Text (15):
In response to a memory read request (other than a READ LOCK, as described below), the memory management unit 25 accesses the primary cache 14 for the read data. If the primary cache 14 determines that requested read data is not present, a "cache miss" or "read miss" condition occurs. In this event, the memory management unit 25 instructs the cache controller unit 26 to continue processing the read. The cache controller unit 26 first looks for the data in the backup cache 15 and fills the block in the primary cache 14 from the backup cache 15 if the data is present. If the data is not present in the backup cache 15, the cache controller unit 26 requests a cache fill on the CPU bus 20 from memory 12. When memory 12 returns the data, it is written to both the Backup cache 15 and to the primary cache 14. The cache controller unit 26 sends four quadwords of data to the memory management unit 25 using instruction-stream cache fill or data-stream cache fill commands. The four cache fill commands together are used to fill the entire primary cache 14 block corresponding to the hexaword (16-word) read address on bus 57. In the case of data-stream fills, one of the four cache fill commands will be qualified with a signal indicating that this quadword fill contains the requested data-stream data corresponding to the quadword address of the read. When this fill is encountered, it will be used to supply the requested read data to the memory management unit 25, instruction unit 22 and/or execution unit 23. If, however, the physical address corresponding to the cache fill command falls into I/O space, only one quadword fill is returned and the data is not cached in the primary cache 14. Only memory data is cached in the primary cache 14.

Detailed Description Text (26):
The CPU 10 initiates the above operations upon memory 12 or the cache of another CPU 28 by transmitting cache coherency commands over the CPU bus 20 and the system bus 10. The cache coherency commands are listed in Table B, together with actions performed when another CPU receives each command.

Detailed Description Text (27):
The instruction read command IREAD requests instructions from an addressed cache block. The DREAD command requests data from an addressed cache block. When intercepted by another CPU, these commands cause no change in the state of the cache of another CPU unless the accessed cache block is owned by another CPU. In this case, the other CPU relinquishes ownership by writing the data in the cache block of its cache back to system memory 12 and setting the cache block of its own cache to a state of "valid-unowned." This kind of writeback-invalidate operation is known as a "Rinval" operation.

Detailed Description Text (28):
The command OREAD requests ownership as well as data from the addressed cache block. The command WRITE transmits data to the system memory 12. If another CPU intercepts either of these commands and has the addressed cache block in its cache, then it invalidates the addressed block in its cache. Moreover, if the other CPU owned the addressed cache block, it gives up ownership and writes back data from the addressed cache block in its cache to the system memory 12. This kind of writeback-invalidate operation is known as an "Oinval" operation.

Detailed Description Text (32):
Upon receipt of a read lock request, the memory management unit 25 always forces a primary cache 14 read miss sequence regardless of whether the referenced data is actually stored in the primary cache. This is necessary in order that the read propagate out to the cache controller unit 26 so that memory lock/unlock protocols can be properly processed. Therefore, the memory management unit 25 transmits a READ LOCK command to the cache controller 26.

Detailed Description Text (33):
Upon receipt of a READ LOCK command, the cache controller obtains ownership of the cache block to be locked, if the cache block is not already owned, before transmitting the referenced data back to the memory management unit 25. Ownership of this interlocked cache block is retained at least until cache controller 26 writes the modified value back into the interlocked cache block upon receipt of a corresponding WRITE UNLOCK command from the memory management unit. Write-back of the block to the system memory 12 is prevented from the time that the cache controller receives the READ LOCK command to the time that the cache controller executes the WRITE UNLOCK command. Moreover, in the preferred system of FIG. 1, once a READ LOCK command has been passed to the cache controller, the cache controller will not process any subsequent data stream read references until the corresponding WRITE UNLOCK command has been executed.

Detailed Description Text (34):
In addition to the READ LOCK and WRITE UNLOCK commands, the memory management unit 25 passes the following commands to the cache controller 26: DREAD (Data Stream Read), READ MODIFY (Data Stream Read with Intent to Write), IPR READ (Internal Processor Read), IREAD (Instruction-stream Read), IPR WRITE (Internal Processor Register Write), and WRITE (Data Write to Memory or I/O Space). In general, the cache controller responds to the DREAD, IREAD, READ MODIFY, and WRITE commands in accordance with the cache coherency protocols described above. The IPR READ and IPR WRITE commands may reference internal registers of the cache controller, and which would not involve access to the cache 15 or the CPU bus 20.

Detailed Description Text (35):
The cache controller responds to an IREAD, DREAD, and READ MODIFY command in a similar fashion by accessing the back-up cache 15, and detecting a "cache hit" if the cache tag matches the requested cache block address and the valid bit of the indexed cache block is set. The back-up cache is accessed in a similar fashion for the READ LOCK command, but a cache bit also requires the ownership bit of the indexed cache block to be set. IREAD and DREAD misses result in IREAD and DREAD commands on the CPU bus 20 and the system bus 11. READ MODIFY, READ.sub.-- LOCK, and WRITE misses result in OREAD commands the CPU bus 20 and the system bus 11.

Detailed Description Text (41):
An invalidate is the mechanism by which the primary cache 14 is kept coherent with

the backup cache 15, and occurs when data is displaced from the backup cache 15 or when backup cache 15 data is itself invalidated. The cache controller unit 26 initiates an invalidate by specifying a hexaword physical address qualified by the Inval command on a bus (59 in FIG. 4). Execution of an Inval command guarantees that the data corresponding to the specified hexaword address will not be valid in the primary cache 14. If the hexaword address of the Inval command does not match to either of the two primary cache 14 tags in the addressed index, no operation takes place. If the hexaword address matches one of the tags, the four corresponding subblock valid bits are cleared to guarantee that any subsequent primary cache 14 accesses of this hexaword will miss until this hexaword is re-validated by a subsequent primary cache 14 fill sequence. A primary cache 14 invalidate operation is interpreted as a NOP (no operation) by the primary cache 14 if the address does not match either tag field in the addressed index.

Detailed Description Text (44):
Both the tags and data for the backup cache 15 are stored in off-chip RAMs, with the size and access time selected as needed for the system requirements. The backup cache 15 may be of a size of from 128K to 2Mbytes, for example. With an access time of 28 nsec, the cache can be referenced in two machine cycles, assuming 14 nsec machine cycle for the CPU 10. The cache controller unit 26 packs sequential writes to the same quadword in order to minimize write accesses to the backup cache. Multiple write commands from the memory management unit 25 are held in an eight-entry write queue (60 in FIG. 4) in order to facilitate this packing, as further described below.

Detailed Description Text (53):
A five-bit command bus 262 from the memory management unit 25 is applied to a controller 306 to define the internal bus activities of the cache controller unit 26. This command bus indicates whether each memory request is one of eight types: instruction stream read, data stream read, data stream read with modify, interlocked data stream read, normal write, write which releases lock, or read or write of an internal or external processor register. These commands affect the instruction or data read latches 299 and 300, or the write packer 301 and the write queue 60. Similarly, a command bus 262 goes back to the memory management unit 25, indicating that the data being transmitted during the cycle is a data stream cache fill, an instruction stream cache fill, an invalidate of a hexaword block in the primary cache 14, or a NOP. These command fields also accompany the data in the write queue, for example.

Detailed Description Text (55):
The CPU bus 20 is a pended, synchronous bus with centralized arbitration. As explained earlier, by "pended" is meant that several transactions can be in process at a given time, rather than always waiting until a memory request has been fulfilled before allowing another memory request to be driven onto the bus 11. The cache controller unit 26 of the CPU 10 may send out a memory read request, and, in the several bus cycles before the system memory 12 sends back the data in response to this request, other memory requests may be driven to the bus 20. The ID (identification) field on the command bus portion of the bus 20 when the data is driven onto the bus 20 specifies which node requested the data, so the requesting node can accept only its own data.

Detailed Description Text (57):
Referring to FIG. 6, the bus 20 consists of a number of lines in addition to the 64-bit, multiplexed address/data lines 20a which carry the addresses and data in alternate cycles as seen in trace (a) of FIG. 5. The lines shared by the nodes on the bus 20 (the CPU 10, the I/O 13a, the I/O 13b and the interface chip 21) include the address/data bus 20a, a four-bit command bus 20b which specifies the current bus transaction during a given cycle (write, instruction stream read, data stream read, etc.), a three-bit ID bus 20c which contains the identification of the bus commander during the address and return data cycles (each commander can have two read transactions outstanding), a three-bit parity bus 20d, and the acknowledge line 20e. All of the command encodings for the command bus 20b and definitions of these transactions are set forth in Table A, below. The CPU also supplies four-phase bus clocks from the clock generator 30 on lines 20f.

Detailed Description Text (58):
In addition to these shared lines in the bus 20, each of the three active nodes CPU 10, I/O 13a and I/O 31b individually has the request, hold and grant lines 20g, 20h and 20i, connecting to the arbiter 325 in the memory interface chip 21. A further function is provided by a suppress line 20j, which is asserted by the CPU 10, for example, in order to suppress new transactions on the bus 20 that the CPU 10 treats as cache coherency transactions. It does this when its two-entry cache coherency queue 61 is in danger of overflowing. During the cycle when the CPU 10 asserts the suppress line 20j, the CPU 10 will accept a new transaction, but transactions beginning with the following cycle are suppressed (no node will be granted command of the bus). While the suppress line 20j is asserted, only fills and writebacks are allowed to proceed from any nodes other than the CPU 10. The CPU 10 may continue to put all transactions onto the bus 20 (as long as WB-only line 20k is not asserted). Because the in-queue 61 is full and takes the highest priority within the cache controller unit 26, the CPU 10 is mostly working on cache coherency transactions while the suppress line 20j is asserted, which may cause the CPU 10 to issue write-disowns on the bus 20. However, the CPU 10 may and does issue any type of transaction while its suppress line 20j is asserted. The I/O nodes 13a and 13b have a similar suppress line function.

Detailed Description Text (59):
The writeback-only or WB-only line 20k, when asserted by the arbiter 325, means that the node it is directed to (e.g., the CPU 10) will only issue write-disown commands, including write disowns due to write-unlocks when the cache is off. Otherwise, the CPU 10 will not issue any new requests. During the cycle in which the WB-only line 20k is asserted to the CPU 10, the system must be prepared to accept one more non-writeback command from the CPU 10. Starting with the cycle following the assertion of WB-only, the CPU 10 will issue only writeback commands. The separate writeback and non-writeback queues 63 and 62 in the cache controller unit 26 of FIG. 4 allow the queued transactions to be separated, so when the WB-only line 20k is asserted the writeback queue 62 can be emptied as needed so that the other nodes of the system continue to have updated data available in system memory 12.

Detailed Description Text (61):
The rules executed by the arbiter 325 are as follows: (1) any node may assert its request line 20g during any cycle; (2) a node's grant line 20i must be asserted before that node drives the bus 20; (3) a driver of the bus 20 may only assert its hold line 20h if it has been granted the bus for the current cycle; (4) if a node has been granted the bus 20, and it asserts hold, it is guaranteed to be granted the bus 20 in the following cycle; (5) hold line 20h may be used in two cases, one to hold the bus for the data cycles of a write, and the other to send consecutive fill cycles; (6) hold must be used to retain the bus for the data cycles of a write, as the cycles must be contiguous with the write address cycle; (7) hold must not be used to retain the bus 20 for new transactions, as arbitration fairness would not be maintained; (8) if a node requests the bus 20 and is granted the bus, it must drive the bus during the granted cycle with a valid command--NOP is a valid command--the CPU 10 takes this a step further and drives NOP if it is granted the bus when it did not request it; (9) any node which issues a read must be able to accept the corresponding fills as they cannot be suppressed or slowed; (10) if a node's WB-only line 20k is asserted, it may only drive the bus 20 with NOP, Read Data Return, Write Disown, and other situations not pertinent here; (11) if a node asserts its suppress line 20j, the arbiter 325 must not grant the bus to any node except that one in the next cycle--at the same time the arbiter must assert the appropriate WB-only lines (in the following cycle, the arbiter must grant the bus normally); (12) the rules for hold override the rules for suppress; (13) the bus 20 must be actively driven during every cycle.

Detailed Description Text (64):
All reads have significant bits in their address down to the quadword (bit <3> of the address). Although fills (which are hexaword in length) may be returned with quadwords in any order, there is a performance advantage if system memory 12 returns the requested quadword first. The bus 20 protocol identifies each quadword using one of the four Read Data Return commands on bus 20b, as set forth in Table C, so that quadwords can be placed in correct locations in backup cache 15 by the cache controller unit 26, regardless of the order in which they are returned. Quadword,

octaword and hexaword writes by the CPU 10 are always naturally aligned and driven onto the bus 20 in order from the lowest-addressed quadword to the highest.

Detailed Description Text (67):
The Bad Write Data command appearing on the bus 20b, as listed in Table C, functions to allow the CPU 10 to identify one bad quadword of write data when a hexaword writeback is being executed. The cache controller unit 26 tests the data being read out of the backup cache 15 on its way to the bus 20 via writeback queue 62. If a quadword of the hexaword shows bad parity in this test, then this quadword is sent by the cache controller unit 26 onto the bus 20 with a Bad Write Data command on the bus 20b, in which case the memory 12 will receive three good quadwords and one bad in the hexaword write. Otherwise, since the write block is a hexaword, the entire hexaword would be invalidated in memory 12 and thus unavailable to other CPUs. Of course, error recovery algorithms must be executed by the operating system to see if the bad quadword sent with the Bad Write Data command will be catastrophic or can be worked around.

Detailed Description Text (68):
As described above, the bus 20 is a 64-bit, pended, multiplexed address/data bus, synchronous to the CPU 10, with centralized arbitration provided by the interface chip 21. Several transactions may be in process at a given time, since a Read will take several cycles to produce the read-return data from the system memory 12 and meanwhile other transactions may be interposed. Arbitration and data transfer occur simultaneously (in parallel) on the bus 20. Four nodes are supported: the CPU 10, the system memory (via bus 11 and interface chip 21) and two I/O nodes 13a and 13b. On the 64-bit bus 20a, data cycles (64-bits of data) alternate with address cycles containing 32-bit addresses plus byte masks and data length fields; a parallel command and arbitration bus carries a command on lines 20b, an identifier field on lines 20c defining which node is sending, and an Ack on line 20e; separate request, hold, grant, suppress and writeback-only lines are provided to connect each node to the arbiter 325.

Detailed Description Text (78):
Referring now to FIG. 7, the interface unit 21 functions to interconnect the CPU bus 20 with the system bus 11. The system bus 11 is a pended, synchronous bus with centralized arbitration. Several transactions can be in progress at a given time, allowing highly efficient use of bus bandwidth. Arbitration and data transfers occur simultaneously, with multiplexed data and address lines. The bus 11 supports writeback caches by providing a set of ownership commands, as discussed above. The bus 11 supports quadword, octaword and hexaword reads and writes to system memory 12. In addition, the bus 11 supports longword-length read and write operations to I/O space, and these longword operations implement byte and word modes required by some I/O devices. Operating at a bus cycle of 64-nsec, the bus 11 has a bandwidth of 125-Mbytes/sec.

Detailed Description Text (79):
The information on the CPU bus 20 is applied by an input bus 335 to a receive latch 336; this information is latched on every cycle of the bus 20. The bus 335 carries the 64-bit data/address, the 4-bit command, the 3-bit ID and 3-bit parity as discussed above. The latch 336 generates a data output on bus 337 and a control output on bus 338, applied to a writeback queue 339 and a non-writeback queue 340, so the writebacks can continue even when non-writeback transactions are suppressed as discussed above. From the writeback queue 339, outputs 341 are applied only to an interface 342 to the system bus 11, but for the non-writeback queue 340 outputs 343 are applied to the interface 342 to the system bus 11 or to an interface 344 to the ROM bus 29. Writebacks will always be going to system memory 12, whereas non-writebacks may be to system memory 12 or to the ROM bus 29. Data received from the system bus 11 at the transmit/receive interface 342 is sent by bus 345 to a response queue 346 as described below in more detail, and the output of this response queue in applied by a bus 347 to a transmit interface 348, from which it is applied to the bus 20 by an output 349 of the interface 348. The incoming data on bus 345, going from system bus 11 to the CPU 10, is either return data resulting from a memory read, or is an invalidate resulting from a write to system memory 12 by another processor 28 on the system bus 11. Incoming data from the ROM bus 29 is applied from the transmit/receive interface 344 by bus 351 directly to the interface

348, without queuing, as the data rate is low on this channel. The arbiter 325 in the interface chip 21 produces the grant signals to the CPU 10 as discussed above, and also receives request signals on line 352 from the transmit interface 348 when the interface 348 wants command of the bus 20 to send data, and provides grant signals on line 353 to grant the bus 20 to interface 348.

Detailed Description Text (83):
The interface chip 21 participates in all bus 20 transactions, responding to Reads and Writes that miss in the backup cache 15, resulting in a system bus 11 Ownership Read operation and a cache fill. The interface chip 21 latches the address/data bus 20a, command bus 20b, ID bus 20c, and parity 20d, into the latch 336 during every bus 20 cycle, then checks parity and decodes the command and address. If parity is good and the address is recognized as being in interface chip 21 space, then Ack line 20e is asserted and the information is moved into holding registers in queues 339 or 340 so that the latches 336 are free to sample the next cycle. Information in these holding registers will be saved for the length of the transaction.

Detailed Description Text (87):
The following CPU 10 generated commands are all treated as a Memory Read by the interface chip 21 (the only difference, seen by the interface chip 21, is how each specific command is mapped to the system bus 11: (1) Memory-space instruction-stream Read hexaword; (2) Memory-space data-stream Read hexaword (ownership); and (3) Memory-space data-stream Read hexaword (no lock or ownership). When any of these Memory Read commands occur on the bus 20 and if the Command/Address parity is good, the interface chip 21 places the information in a holding register.

Detailed Description Text (90):
For CPU Memory Write operations, the following four CPU 10 generated commands are treated as Memory Writes by the interface chip 21 (the only difference, seen by the interface chip 21, is how each specific command is mapped to the system bus 11: (1) Memory-space Write Masked quadword (no disown or unlock); (2) Memory-space Write Disown quadword; (3) Memory-space Write Disown hexaword; and (4) Memory-space Bad Write Data hexaword.

Detailed Description Text (91):
For deallocates due to CPU Reads and Writes, when any CPU 10 tag lookup for a read or a write results in a miss, the cache block is deallocated to allow the fill data to take its place. If the block is not valid, no action is taken for the deallocate. If the black is valid but not owned, the block is invalidated. If the block is valid and owned, the block is sent to the interface chip 21 on the bus 20 and written back to system memory 12 and invalidated in the tag store. The Hexaword Disown Write command is used to write the data back. If a writeback is necessary, it is done immediately after the read or write miss occurs. The miss and the deallocate are contiguous events for the cache controller and are not interrupted for any other transaction.

Detailed Description Text (92):
For Read-Lock and Write-Unlock operations, the CPU 10 cache controller unit 26 receives Read Lock/Write Unlock pairs from the memory management unit 25; it never issues those commands on the bus 20, but rather uses Ownership Read-Disown Write instead and depends on use of the ownership bit in system memory 12 to accomplish interlocks. A Read lock which does not produce an owned hit in the backup cache 15 results in an ORead on the bus 20, whether the cache 15 is on or off. When the cache is on, the Write Unlock is written into the backup cache 15 and is only written to memory 12 if requested through a coherence transaction. When the cache 15 is off, the Write Unlock becomes a Quadword Disown Write on the bus 20.

Detailed Description Text (95):
The invalidate queue 355 is twelve entries deep in the example. The interface chip 21 uses the system bus 11 suppress line to suppress bus 11 transactions in order to keep the responder queue 355 from overflowing. If (for example) ten or more entries in the responder 355 queue are valid, the interface chip 21 asserts the suppress line to system bus 11. Up to two more bus 11 writes or three bus 11 reads can occur once the interface chip 21 asserts the suppress signal. The suppression of system bus 11 commands allows the interface chip 21 and CPU 10 cache controller unit 26 to

catch up on invalidate processing and to open up queue entries for future invalidate addresses. When the number of valid entries drops below nine (for example), the interface chip 21 deasserts the suppress line to system bus 11.

Detailed Description Text (96):
A potential problem exists if an invalidate address is received which is in the same cache subblock as an outstanding cacheable memory read. The cache controller unit 26 tag lookup will produce a cache miss since that subblock has not yet been validated. Since the system bus 11 request that generated this invalidate request may have occurred after the command cycle went on the system bus 11, this invalidate must be processed. The CPU 10 cache controller unit 26 maintains an internal state which will force this cache subblock to be invalidated or written back to memory once the cache fill completes. The cache controller unit 26 will process further invalidates normally while waiting for the cache fill to complete.

Detailed Description Text (102):
The approach of FIGS. 8 and 9 has several advantages over the use of a single queue, without greatly increasing the complexity of the design. The advantages all pertain to providing the necessary performance, while reducing the chip size. The specific main advantages are: (1) The same performance obtained with a large, unified queue can be realized with far less space using the split queue method; (2) Each queue can be earmarked for a specific type of data, and there can be no encroaching of one data type into the other. As such, the two types of queues (invalidate and return data) can be tuned to their optimum size. For example, the invalidate queue might be seven (small) slots while the read data queue might be five or six (large) slots. This would provide a smooth read command overlap, while allowing invalidates to be processed without unduly suppressing the system bus 11; (3) The read data queue 356 can be increased to accommodate two outstanding reads without worrying about the size of the invalidate queue, which can remain the same size, based upon its own needs.

Detailed Description Text (105):
As introduced above, the fill CAM 302 in FIG. 4 holds addresses of outstanding misses to the back-up cache 15. By accessing the fill CAM before accessing the back-up cache 15, further access to the missed cache block for another memory management unit command or a cache coherency transaction is stalled until the fill is completed. When the cache is off or in ETM, however, writes are not checked for block conflict, but are sent immediately to memory.

Detailed Description Text (107):
A miss to a cache block in the back-up cache 15 is outstanding until the fill data has been received from the system memory 12. When a read transaction is issued to the system memory 12 to request the fill data, the fill CAM entry is validated by setting the valid bit, the address field is loaded, and the appropriate status bits RDLK, IREAD, OREAD, WRITE, and TO.sub.-- MBOX are set depending on the particular command, from the memory management unit, that required the access to the back-up cache 15. RIP, OIP, RDLK.sub.-- FL.sub.-- DONE, and REQ.sub.-- FILL.sub.-- DONE are cleared. If the cache is off, in ETM, or the miss is for an I/O reference, DNF is set; otherwise, it is cleared. COUNT is set to zero if four fill quadwords are expected; it is set to 3 if only one quadword is expected.

Detailed Description Text (108):
The fill CAM status bits are set under certain conditions upon the return of fill data or cache coherency commands from the CPU bus that are associated with the miss address in the fill CAM. If an abort request arrives from the memory management unit 25, and the entry is marked IREAD, then the TO.sub.-- MBOX bit is cleared. When the data returns in this case, it will be written into the back-up cache (if DNF is not set), but it will not be sent to the memory management unit 25.

Detailed Description Text (114):
When the cache controller 26 receives a READ LOCK command from the memory management unit 25, further access to the cache block specified by the READ LOCK command must be stalled until the corresponding WRITE UNLOCK command is received and executed by the cache controller 26, as was introduced above. One way to perform this function would be to store the address of the outstanding read lock in a separate register,

and to check the address in this register of any new memory access command from the memory management unit 25 or cache coherency transaction from the CPU bus 20; if the address matched, that command or transaction would be stalled until the corresponding WRITE LOCK command would be executed. In the cache controller 26 of FIG. 4, however, the fill CAM 302 is used to obtain the same result. The primary purpose of the fill CAM 302 is to hold the addresses and other information related to memory access commands which have missed in the back-up cache so that further accesses to those cache blocks can be prevented until the cache fills are returned from memory. But the fill CAM 302 is also used to hold outstanding READ LOCK information, so that access to a locked cache block is also prevented until the corresponding WRITE UNLOCK is executed.

Detailed Description Text (115):
In a preferred arrangement, when the cache controller 26 receives a READ LOCK command from the memory management unit 25, the cache controller places the block address specified by the READ LOCK command into an entry of the fill CAM 302, regardless of whether or not the block address hits in the back-up cache 15. At the same time, the following control bits are set in that fill CAM entry: RDLK (to indicate that a READ LOCK is in progress); OREAD (to indicate that the READ LOCK is an Ownership-Read type of transaction); TO.sub.-- MBOX (if the returning fill data is to be sent to the memory management unit 25); and VALID (to indicate that the entry is currently valid).

Detailed Description Text (118):
When processing by the cache controller 26 terminates in an error, information related to the fill CAM 302 is available from a pair of registers 307, 308 (FIG. 4) that can be accessed by the execution unit via IPR READ commands transmitted by the memory management unit 25 to the cache controller 26. Error information is also available from internal processor status registers 309, 310, and 311 which store address information from the internal address bus 288, data from the internal data bus 289, and data from the bus 292, respectively. Therefore, when an error condition arises in the cache controller 26, an error signal is sent to the micro-controller 24 (FIG. 1) of the execution unit 23, permitting the micro-controller to execute an error handling sequence which may access the internal processor registers of the cache controller.

Detailed Description Text (119):
In the case of a READ LOCK error, the execution unit microcode may resume normal instruction execution by causing the memory management unit 25 to send an IPR WRITE command to the CEFSTS 308 (Fill CAM Error Register) which has the side effect of clearing any set RDLK bits and VALID bits in the fill CAM 302. Therefore, the cache controller 26 becomes free to resume processing of commands from the memory management unit 25 and cache coherency transactions from the CPU bus 20.

Detailed Description Text (123):
OREAD indicates that the transaction in error was an OREAD; the OREAD may have been done for a WRITE, a READ LOCK, or a READ MODIFY command.

Detailed Description Text (124):
WRITE indicates that the transaction in error was an OREAD done because of a WRITE command.

Detailed Description Text (141):
As described above with reference to FIG. 4, a data stream read request received by the cache controller 26 from the memory management unit 25 is held in a data read latch 299. This D-read latch 299 is one entry deep and holds the address of the data stream read request and a five-bit code indicating the specific read command. The data stream read requests include DREAD, READ MODIFY, READ LOCK, and IPR READ commands.

Detailed Description Text (142):
An IREAD command received by the cache controller unit 26 from the memory management unit 25 is held in an instruction read latch 300. This I-read latch 300 is one entry deep and holds the address of the IREAD command, together with a five-bit code for the IREAD command.

Detailed Description Text (144):
The write packer accumulates memory-space writes to the same quadword which arrive sequentially, so that only one write has to be done into the back-up cache 15. Only WRITE commands to memory space to the same quadword are packed together. When a memory space WRITE command to a different quadword is received, the write packer 301 is flushed by transferring its contents into the write queue 60. Other kinds of write requests pass immediately from the write packer 301 into the write queue 60 after the write packer 301 is flushed by transferring any existing data into the write queue. The write packer 301 is also flushed if an IREAD or DREAD arrives specifying the same hexaword as that of the entry in the write packer. The write packer 301 is also flushed whenever any condition for flushing the write queue, as described below, is met on the entry in the write packer. Moreover, the execution unit (23 in FIG. 1) can write to a control register to set a "disable pack" bit so that every write passes directly through the write packer without delay.

Detailed Description Text (147):
When a data read request is received in the D-read latch 299, its hexaword address is compared to the write addresses in the write packer 301 and in all entries in the write queue 60. Any entry with a matching hexaword address has its corresponding DWR conflict bit set. The DWR conflict bit is also set if the write packer or the write queue entry is an IPR WRITE command, a WRITE UNLOCK command, or an I/O space write. If any DWR conflict bit is set, the write queue 60 takes priority over the data read request allowing the writes up to the point of the conflicting write to execute first.

Detailed Description Text (148):
In a similar fashion, when an instruction read is received in the I-read latch 300, its hexaword address is compared to the write addresses in the write packer 301 and in all entries in the write queue 60. Any entry with a matching hexaword address has its corresponding IWR conflict bit set. The IWR conflict bit is also set if the write packer or the write queue entry is an IPR WRITE command, a WRITE UNLOCK command, or an I/O space write. If any IWR conflict bit is set, the write queue takes priority over instruction reads, allowing the writes up to the point of the conflicting write to execute first.

Detailed Description Text (149):
All of the DWR conflict bits are OR'd together to make one signal which is sent to the C-box controller 306 to indicate that a write conflict exists on the current entry of the D-read latch 299. Similarly, all of the valid IWR conflict bits are OR'd together to make one signal which is sent to the C-box controller 306 to indicate that a write conflict exists on the current entry of the I-read latch 300. The controller 306 uses these signals to decide how to prioritize the execution of the commands currently in the D-read latch 299, I-read latch 300, and write queue 60.

Detailed Description Text (154):
In particular, when a READ LOCK command arrives from the memory management unit 25, DWR conflict bits for all valid entries in the write packer 301 and the write queue 60 are set so that all writes preceding the READ LOCK are done before the READ LOCK is done. When any IPR READ command arrives from the memory management unit 25, all DWR conflict bits for valid entries in the write packer 301 and the write queue 60 are set, so that previous writes complete first. When any instruction stream I/O space read arrives, all IWR conflict bits for valid entries in the write packer 301 and the write queue 60 are set, so that previous writes complete first.

Detailed Description Text (158):
Thus, all memory access commands from the Mbox, except memory space reads and writes, unconditionally force the flushing of the WRITE.sub.-- QUEUE (the completion of all entries marked with a conflict bit). A memory space read causes a flush only up through conflicting previous memory space writes.

Detailed Description Text (165):
As shown in FIG. 11, an entry of the write queue 60 includes a valid bit, a data-stream write-read conflict bit DWR, an instruction-stream write-read conflict

bit IWR, a five-bit command (CMD) indicating a specific command from the memory management unit (25 in FIG. 1), a thirty-two bit physical address, eight byte enable bits enabling respective bytes of a quadword to be written, and the quadword of data to be written.

Detailed Description Text (173):
The presence of a READ LOCK, and IPR READ, and a D-stream I/O command is detected by decoding logic 458. AND gates 459, 460 set the DWR conflict bit when the entry is valid and such a command just occurs during the current clock cycle (when the signal NEW D-READ is asserted), so long as the entry is not also removed at the end of the current clock cycle.

Detailed Description Text (174):
The presence of an I/O space write, an IPR WRITE, or a WRITE UNLOCK in the entry is detected by decoding logic 461. AND gates set the DWR conflict bit when the entry is valid and a D-read command just occurs during the current clock cycle (when the signal NEW D-READ is asserted), so long as the entry is not also removed at the end of the current cycle. To eliminate the decoding logic 461, however, the command codes for the write commands could be selected so that the presence of such a command is indicated by the state of a particular one of the five command bits CMD.

Detailed Description Text (178):
Turning now to FIG. 17, the C-box controller 306 includes a C-box arbiter 471, an M-box interface control 472, a tag store control 473, a data RAM control 474, and a CPU bus interface control 475. The arbiter 471 arbitrates among simultaneous requests for service including memory access commands from the memory management unit (25 in FIG. 1) and cache coherency commands from the CPU bus (20 in FIG. 1), as further described below with reference to FIG. 18. A request granted priority is given access to the internal address bus (288 in FIG. 4) of the back-up cache controller, and executed by initiating tasks performed by the tag store control 473 and the back-up cache control 474, as further described below with reference to FIGS. 19 and 20.

Detailed Description Text (179):
The M-box interface control 472 controls the receipt of commands from the memory management unit 25 into the read latches 299, 300 and the write packer 301, and sending data and invalidates from the out latch 296 to the memory management unit (See also FIG. 4).

Detailed Description Text (187):
In step 485, the arbiter gives the next highest priority to a read lock in progress. A read lock in progress is indicated by either one of the RDLK bits in the fill CAM 302 being set. When a read lock is in progress, the write queue is inspected in step 486. If the write queue is empty, as indicated by its removal pointer pointing to an entry having its valid bit clear, the arbiter is finished arbitrating for the current cycle. Otherwise, in step 487, the write queue is serviced. In particular, the WRITE UNLOCK corresponding to the READ LOCK is the only write command which will be received and loaded into the write queue unless an error occurs. When an error occurs, an IPR WRITE command will be serviced from the write queue, causing the RDLK bit in the fill CAM to be cleared.

Detailed Description Text (191):
Turning now to FIG. 19, there is shown a flow chart of the steps followed by the arbiter 471 in servicing the D-read latch 299, the I-read latch 300, and the write queue 60. In steps 501, 502, 503, the source given priority asserts the address of its memory command upon the internal address bus 288 of the cache controller (see FIG. 4). If the memory command accesses an internal processor register (IPR) or I/O or a write unlock, as tested in step 504, then the command is completed in step 505. (To simplify implementation, the test in step 504 can be done concurrently with step 506 so that the fill CAM is always addressed and a hit always causes execution of a command other than a WRITE UNLOCK to stall.) If, however, the memory command accesses memory space, then in step 506, processing of the task is halted if there is a hit in the fill CAM. In this case, the memory space access conflicts with an outstanding fill or an outstanding READ LOCK. If, however, there is not a conflict with an outstanding fill or READ LOCK, then in step 507, execution branches

depending on whether the cache is in the above-described error transition mode or whether the memory access is requested by an ownership command. If so, then in step 508, the tag RAMs are accessed to determine whether there is a cache hit in an owned block. If not, then if the cache is in the error transition mode, as tested in step 509, the back-up cache is bypassed and the read or write is sent directly to system memory (12 in FIG. 1) in step 510. If, however, in step 509, the cache was not operating in the error transition mode, then in step 511, an ownership read is sent to memory, and in step 512, the fill CAM is set to record that the refill is in progress. Moreover, if the addressed block in the cache is owned, as tested in step 513, then in step 514, the cache block is de-allocated and written back to memory in the next task. In other words, in step 514, a flag is set which is inspected by the arbitrator in step 481 of FIG. 18 to determine whether a need to de-allocate was caused by the previous task.

Detailed Description Text (192):
If in step 508 there was a cache hit in an owned block of the back-up cache, then in step 515, execution branches depending on whether the cache is in the error transition mode. If so, then in step 516, an ownership transaction is sent to memory, and the memory block is de-allocated and written back to memory in the next task. From step 515 or 516, execution continues in step 517 to complete the command.

Detailed Description Text (193):
If in step 507 it was found that the cache was neither in the error transition mode nor the command was an ownership command, then in step 518, execution branches depending on whether there was a cache hit. If so, then the command is completed in step 517. If not, then execution branches to step 519, where a refill of the cache block is begun by sending a data read or instruction read to memory. The fact that the refill is in progress is recorded in the fill CAM in step 512, and if the address block in the cache is owned, as tested in step 513, then in step 514, the address block is de-allocated and written back to memory in the next task.

Detailed Description Text (198):
It should be appreciated that the control sequences in FIGS. 18-20B assume that various resources are available in the back-up cache controller for performing a selected task. If the required resources are not available, then a next lowest priority task may be performed if resources are available for performing that next-lowest priority task. In particular, the necessary conditions before servicing a fill from the in queue 61 are: (1) the data RAMs and the tag store must be free, and (2) if RIP or OIP is set in either fill CAM entry, the write-back queue 63 must not be full, because a write-back may be necessary at the completion of the fill. Necessary conditions before servicing a cache coherency request from the in-queue 61 are that the tag store must be free. If the cache coherency request hits owned and requires a write-back, and the write-back queue 63 is full, then the cache coherency request is stalled until the write-back queue is no longer full. Necessary conditions before servicing a command from the D-read latch 299 or the I-read latch 300 are: (1) the data RAMs and the tag store must be free; (2) a fill CAM entry must be available, in case the read misses; (3) there must be an available entry in the non-write-back queue 62, in case the read misses; (4) there must be no valid entry in the fill CAM for the same cache block as that of the new request; (5) there must be no RDLK bits set in the fill CAM, indicating that a READ LOCK is in progress; and (6) there must be no block conflict with any write queue entry. If a read misses owned and requires a de-allocate, and the write-back queue 63 is full, then the read is stalled until the write-back queue is no longer full. Necessary conditions before servicing a full quadword write from the write queue 60 are: (1) the tag store must be free; (2) a fill CAM entry must be available, in case the write misses and requires an OREAD; (3) there must be an available entry in the non-write-back queue 62, in case the write misses; (4) there must be no valid entry in the fill CAM for the same cache block as that of the new request; and (5) if there is a READ LOCK in the fill CAM, the fills for the READ LOCK must have completed. If the full quadword write misses owned and requires a deallocate, and the write-back queue 63 is full, the quadword write is stalled until the WRITE BACK queue is no longer full.

Detailed Description Paragraph Table (2):
TABLE B _____ Normal Backup Cache Behavior cache

state of the block in the cache coherency Invalid Valid, valid, command block unowned block owned block IREAD, no action no action writeback, set block DREAD state to valid-unowned OREAD, no action invalidate writeback, invalidate WRITE WDISOWN no action no action no action

Detailed Description Paragraph Table (3):

TABLE C CPU Bus Command Encodings and Definitions Command Bus Field Abbrev. Transaction Type Function 0000 NOP No Nop No Operation Operation 0010 WRITE Write Addr Write to memory with byte enable if quadword or octaword 0011 WDISOWN Write Addr Write memory; Disown cache disowns block and returns ownership to memory 0100 IREAD Instruction Addr Instruction- Stream stream read Read 0101 DREAD Data Addr Data-stream Stream read (without Read ownership) 0110 OREAD D-Stream Addr Data-stream Read read claiming Ownership ownership for the cache 1001 RDE Read Data Data Used instead Error of Read Data Return in the case of an error. 1010 WDATA Write Data Data Write data Cycle is being transferred 1011 BADWDATA Bad Write Data Write data with Data error is being transferred 1100 RDRO Read Data0 Data Read data is Return(fill) returning corresponding to QW 0 of a hexaword. 1101 RDR1 Read Data1 Data Read data is Return(fill) returning corresponding to QW 1 of a hexword. 1110 RDR2 Read Data2 Data Read data is Return(fill) returning corresponding to QW 2 of a hexaword. 1111 RDR3 Read Data3 Data Read data Return(fill) is returning corresponding to QW 3 of a hexaword.

WEST

Generate Collection Print

L14: Entry 7 of 13 File: USPT Apr 4, 1995

DOCUMENT-IDENTIFIER: US 5404483 A
TITLE: Processor and method for delaying the processing of cache coherency transactions during outstanding cache fills

Abstract Text (1):
A processor and method for delaying the processing of cache coherency transactions during outstanding cache fills in a multi-processor system using a shared memory. A first processor fetches data having a specified address by addressing a cache memory, and when the specified address is not in the cache, saving the specified address in a fill address memory, and sending a fill request to the shared memory. Before return of fill data, the first processor receives a cache coherency request including the specified address from a second processor requesting invalidation of an addressed block of data. The first processor responds by checking whether the fill address memory includes the specified address, and upon finding the specified address in the fill address memory, delaying execution of the cache coherency request until the fill data is returned, and when the fill data is returned, using the fill data without retaining a validated block of the fill data in the cache. In a preferred embodiment, the fill memory is a content-addressable memory including a plurality of entries, and each entry has a fill address, an ownership fill bit (OREAD), an ownership-read invalidate pending bit (OIP), and a read invalidate pending bit (RIP). The OIP or RIP bit is set when execution of a cache coherency request is delayed, and these bits are read upon completion of a fill to execute the delayed request.

Parent Case Text (2):
The present application is a continuation-in-part of Ser. No. 07/547,699, filed Jun. 29, 1990, entitled BUS PROTOCOL FOR HIGH-PERFORMANCE PROCESSOR, by Rebecca L. Stamm et al., now abandoned in favor of continuation application Ser. No. 08/034,581, filed Mar. 22, 1993, entitled PROCESSOR SYSTEM WITH WRITEBACK CACHE USING WRITEBACK AND NON WRITEBACK TRANSACTIONS STORED IN SEPARATE QUEUES, by Rebecca L. Stamm, et al., issued on May 31, 1994, as U.S. Pat. No. 5,317,720, and Ser. No. 07/547,597, filed Jun. 29, 1990, entitled ERROR TRANSITION MODE FOR MULTI-PROCESSOR SYSTEM, by Rebecca L. Stamm et al., issued on Oct. 13, 1992, as U.S. Pat. No. 5,155,843, incorporated herein by reference. The present application is related to Stamm et al., "Preventing Access to Locked Memory Block By Recording Lock in Content Addressable Memory with Outstanding Cache Fills," Ser. No. 07/902,122, filed Jun. 22, 1992, concurrently with the present application.

Brief Summary Text (4):
This invention is directed to digital computers, and more particularly to cache coherency transactions in a multi-processor system following a cache ownership protocol. Specifically, the invention relates to maintaining cache coherency in such a system having a "pended" bus permitting cache coherency transactions to be transmitted among processors during outstanding fills.

Brief Summary Text (6):
Processors in a multi-processor computer system typically communicate via a shared memory. To improve system performance, each processor has a cache memory for temporarily storing copies of data being accessed. Such a hierarchical memory system may follow either a "write through" or a "write back" protocol. In a "write through" protocol, a processor immediately writes data to the shared memory so that any other processor may fetch the most recent memory state from the shared memory. In a

"writeback" protocol, a processor writes data to its cache, but this new memory state is written back to the shared memory only when the memory space in the cache needs to be used for different addresses in a cache fill operation, or when another processor needs the new memory state. Therefore the writeback protocol reduces the number of memory access operations to the shared memory when the new memory state is not needed by the other processors. In general, the write through protocol is preferred when the different processors frequently access the same shared memory addresses, and the write back protocol is preferred when the different processors infrequently access the same shared memory addresses.

Brief Summary Text (7):
Whenever processors communicate via a shared memory, it is desirable to require the processors to follow a protocol ensuring that a memory address is not written to simultaneously by more than one processor, or else the result of one processor will be nullified by the result of another processor. Such synchronization of memory access is commonly achieved by requiring a processor to obtain an exclusive privilege to write to an addressed portion of the shared memory, before executing a write operation. In a multi-processor system employing writeback caches, such an exclusive privilege gives rise to a cache coherency problem in which data written in the cache of a processor having such an exclusive privilege might be the only valid copy of data for the addressed portion of memory. A cache coherency protocol is required which permits a processor to obtain readily the valid copy of data as well as the privilege to write to it.

Brief Summary Text (8):
One known cache coherency protocol for a multi-processor system employing writeback caches is based on the concept of block ownership; an addressed portion of memory the size of a cache block is either owned by the shared memory or it is owned by one of the writeback caches. Only one of the processors, or the shared memory, may own the block of memory at any given time, and this ownership is indicated by an ownership bit for each block in the shared memory and in each of the caches. A processor may write to a block only when the processor owns the block. Therefore the ownership bits always identify a unique "valid" block in the system. Shared read-only access to a block is permitted only when the shared memory owns the block. To indicate whether a processor may read a block, each of the caches includes, for each block, a "valid" bit. When a processor desires to read a block that is not valid in its cache, it issues a read transaction to the shared memory, requesting the shared memory to fill its cache with valid data. When a processor desires to write to a block which it does not own, it issues an ownership-read transaction to the shared memory, requesting ownership as well as a fill. From the perspective of the other processors, these transactions are cache coherency transactions, which request any other processor having ownership to give up ownership and writeback the data of the requested block, and in the case of an ownership read transaction, further request the other processors to invalidate any copies of the requested block.

Brief Summary Text (9):
Typically the time for a cache coherency transaction to be transmitted over a system bus is much shorter than the time for fill data to be retrieved from the shared memory. Therefore system performance can be improved by use of a pended bus (i.e., a bus which permits more than one transaction to be pending on the bus at any given time). The use of such a "pended" bus, however, leads to a problem of data coherency where a processor may issue a read transaction to fill a cache block, but before receiving the fill data, the processor may receive a cache coherency transaction requesting the cache block to be disowned or invalidated. A conventional way of handling this problem is for the processor to immediately invalidate the cache block by clearing the "valid" bit, which effectively discards the fill data when it is received. But in this case the processor which first requested the data does not get to use it, and must reissue a read transaction to get the data.

Brief Summary Text (11):
In accordance with a basic aspect of the invention, there is provided a method of operating a first processor in a multi-processor system that has a second processor and a shared memory accessed by both of the first and second processors. The first processor has a cache memory for storing blocks of data and associated addresses.

The first processor fetches data having a specified address by addressing the cache memory with the specified address, and when the specified address is not found in the cache memory, saving the specified address in a fill address memory, and sending a fill data request including the specified address to the shared memory. Before receipt of the requested fill data from the shared memory, the first processor receives a cache coherency request including the specified address from the second processor requesting invalidation of a block of data having the specified address. The first processor responds to the cache coherency request by checking whether the fill memory includes the specified address, and upon finding that the fill memory includes the specified address, delaying execution of the cache coherency request until the fill data is received by the first processor, and when the fill data is received by the first processor, using the fill data without retaining a validated block of the fill data in the cache.

Brief Summary Text (12):
In a preferred embodiment, the fill memory is a content-addressable memory (CAM) including a plurality of entries, and each of the entries has a fill address, and associated with the fill address, an ownership fill bit (OREAD), an ownership-read invalidate pending bit (OIP), and a read invalidate pending bit (RIP). The ownership fill bit (OREAD) is set when the first processor requests a fill with ownership of a block of fill data having the associated fill address. The ownership-read invalidate pending (OIP) bit is set when the first processor receives from the second processor a request for ownership of the block of data having the associated fill address. The read invalidate pending (RIP) bit is set when the first processor receives from the second processor a request to read the block of data having the associated fill address, and the ownership fill bit (OREAD) associated with the fill address has been set. Upon receipt of a block of fill data from the shared memory, the first processor loads the block of fill data into the cache memory and subsequently uses the fill data for a data processing operation. Immediately after loading the block of fill data into the cache memory, however, the block of fill data is invalidated when the associated OIP bit has been set, and the block of fill data is disowned and written back to the shared memory when the associated OREAD bit has been set and the associated OIP or RIP bit has been set. Finally, the first processor clears the entry in the fill memory having the address of the block of fill data.

Drawing Description Text (3):
FIG. 1 is a block diagram of a multi-processor computer system incorporating the present invention;

Detailed Description Text (2):
The Multi-Processor System

Detailed Description Text (3):
Referring to FIG. 1, according to one embodiment, a multi-processor computer system employing features of the invention includes a central processing unit (CPU) chip or module 10 connected by a system bus 11 to a system memory 12, an input/output (I/O) unit 13c, and to additional CPU's 28. As will be further described below with reference to FIG. 6, two I/O units 13a, 13b may also be connected directly to a bus 20. In a preferred embodiment the CPU 10 is formed on a single integrated circuit, although the present invention may be used with a CPU implemented as a chip set mounted on a single circuit board or multiple boards.

Detailed Description Text (17):
The present invention more particularly concerns the operation of the cache-controller 26 and maintenance of coherency of the back-up cache 15 with the memory 12 and caches of the other CPU's 28 in the multi-processor system in FIG. 1. Therefore, the specific construction of the components in the CPU 10 other than the cache controller 26 are not pertinent to the present invention. The reader, however, may find additional details in the above-referenced U.S. application Ser. No. 07/547,597, filed Jun. 29, 1990, and issued on Oct. 13, 1992, as U.S. Pat. No. 5,155,843, incorporated herein by reference.

Detailed Description Text (19):
Cache coherency in the multi-processor system of FIG. 1 is based upon the concept of ownership; a hexaword (16-word) block of memory may be owned either by the system

memory 12 or by a backup cache 15 in a CPU on the bus 11--in a multi-processor system. Only one of the caches, or system memory 12, may own the hexaword block at a given time, and this ownership is indicated by an ownership bit for each hexaword in both memory 12 and the backup cache 15 (1 for own, 0 for not-own) .

Detailed Description Text (20):
Shared read-only access to a block among the CPUs 10, 28 is permitted only when system memory 12 owns the block. A CPU may write to a block only when the CPU owns the block. These rules ensure that there is always a unique "valid" block of data in the system, identified by the ownership bits in the caches, and a CPU will always read data from the valid block and write data to the valid block.

Detailed Description Text (21):
The multi-processor system follows certain protocols which ensure rapid access to the valid data of an addressed block. Each back-up cache 15 maintains two bits associated with each cache block. These two bits are called VALID and OWNED, and they determine the state of the cache block as shown in TABLE A.

Detailed Description Text (30):
The preferred embodiment of FIG. 1, however, does have one instance where one CPU will not immediately relinquish ownership to another CPU. The preferred embodiment executes (VAX) instructions, including certain "interlocked" instructions that are guaranteed to perform atomic operations upon memory in a multi-processing environment. An example is an "add aligned word interlocked" instruction (ADAWI) which adds a first operand to a second operand and sets the second operand to the sum. The destination operand has an access type of "modify" raising the possibility that one CPU might obtain ownership of a cache block between the time that the second operand is read from memory and the time that the second operand is modified and written back to memory, leading to a result in memory which might not appear consistent under certain program sequences. Computers which execute (VAX) instructions in a multi-processing environment typically prevent such an interruption of memory access by using the execution unit to request fetching of the second operand and to request a memory "read lock" when fetching the second operand from memory, and to request a memory "write unlock" when putting the result back to memory.

Detailed Description Text (40):
The primary cache 14 must always be a coherent cache with respect to the backup cache 15. The primary cache 14 must always contain a strict subset of the data cached in the backup cache 15. If cache coherency were not maintained, incorrect computational sequences could result from reading "stale" data out of the primary cache 14 in multi-processor system configurations.

Detailed Description Text (57):
Referring to FIG. 6, the bus 20 consists of a number of lines in addition to the 64-bit, multiplexed address/data lines 20a which carry the addresses and data in alternate cycles as seen in trace (a) of FIG. 5. The lines shared by the nodes on the bus 20 (the CPU 10, the I/O 13a, the I/O 13b and the interface chip 21) include the address/data bus 20a, a four-bit command bus 20b which specifies the current bus transaction during a given cycle (write, instruction stream read, data stream read, etc.), a three-bit ID bus 20c which contains the identification of the bus commander during the address and return data cycles (each commander can have two read transactions outstanding), a three-bit parity bus 20d, and the acknowledge line 20e. All of the command encodings for the command bus 20b and definitions of these transactions are set forth in Table A, below. The CPU also supplies four-phase bus clocks from the clock generator 30 on lines 20f.

Detailed Description Text (58):
In addition to these shared lines in the bus 20, each of the three active nodes CPU 10, I/O 13a and I/O 31b individually has the request, hold and grant lines 20g, 20h and 20i, connecting to the arbiter 325 in the memory interface chip 21. A further function is provided by a suppress line 20j, which is asserted by the CPU 10, for example, in order to suppress new transactions on the bus 20 that the CPU 10 treats as cache coherency transactions. It does this when its two-entry cache coherency queue 61 is in danger of overflowing. During the cycle when the CPU 10 asserts the

suppress line 20j, the CPU 10 will accept a new transaction, but transactions beginning with the following cycle are suppressed (no node will be granted command of the bus). While the suppress line 20j is asserted, only fills and writebacks are allowed to proceed from any nodes other than the CPU 10. The CPU 10 may continue to put all transactions onto the bus 20 (as long as WB-only line 20k is not asserted). Because the in-queue 61 is full and takes the highest priority within the cache controller unit 26, the CPU 10 is mostly working on cache coherency transactions while the suppress line 20j is asserted, which may cause the CPU 10 to issue write-disowns on the bus 20. However, the CPU 10 may and does issue any type of transaction while its suppress line 20j is asserted. The I/O nodes 13a and 13b have a similar suppress line function.

Detailed Description Text (68):
As described above, the bus 20 is a 64-bit, pended, multiplexed address/data bus, synchronous to the CPU 10, with centralized arbitration provided by the interface chip 21. Several transactions may be in process at a given time, since a Read will take several cycles to produce the read-return data from the system memory 12 and meanwhile other transactions may be interposed. Arbitration and data transfer occur simultaneously (in parallel) on the bus 20. Four nodes are supported: the CPU 10, the system memory (via bus 11 and interface chip 21) and two I/O nodes 13a and 13b. On the 64-bit bus 20a, data cycles (64-bits of data) alternate with address cycles containing 32-bit addresses plus byte masks and data length fields; a parallel command and arbitration bus carries a command on lines 20b, an identifier field on lines 20c defining which node is sending, and an Ack on line 20e; separate request, hold, grant, suppress and writeback-only lines are provided to connect each node to the arbiter 325.

Detailed Description Text (70):
The backup cache 15 for the CPU 10 is a "write-back" cache, so there are times when the backup cache 15 contains the only valid copy of a certain block of data, in the entire multi-processor system of FIG. 1. The backup cache 15 (both tag store and data store) is protected by ECC. Check bits are stored when data is written to the cache 15 data RAM or written to the tag RAM, then these bits are checked against the data when the cache 15 is read, using ECC check circuits 330 and 331 of FIG. 4. When an error is detected by these ECC check circuits, an Error Transition Mode is entered by the C-box controller 306; the backup cache 15 can't be merely invalidated, since other system nodes 28 may need data owned by the backup cache 15. In this error transition mode, the data is preserved in the backup cache 15 as much as possible for diagnostics, but operation continues; the object is to move the data for which this backup cache 15 has the only copy in the system, back out to system memory 12, as quickly as possible, but yet without unnecessarily degrading performance. For blocks (hexawords) not owned by the backup cache 15, references from the memory management unit 25 received by the cache controller unit 26 are sent to system memory 12 instead of being executed in the backup cache 15, even if there is a cache hit. For blocks owned by the backup cache 15, a write operation by the CPU 10 which hits in the backup cache 15 causes the block to be written back from backup cache 15 to system memory 12, and the write operation is also forwarded to system memory 12 rather than writing to the backup cache 15; only the ownership bits are changed in the backup cache 15 for this block. A read hit to a valid-owned block is executed by the backup cache 15. No cache fill operations are started after the error transition mode is entered. Cache coherency transactions from the system bus 20 are executed normally, but this does not change the data or tags in the backup cache 15, merely the valid and owned bits. In this manner, the system continues operation, yet the data in the backup cache 15 is preserved as best it can be, for later diagnostics.

Detailed Description Text (80):
Referring to FIG. 8, the response queue 346 employs separate queues 355 and 356 for the invalidates and for return data, respectively. The invalidate queue 355 may have, for example, twelve entries or slots 357 as seen in FIG. 9, whereas the return data queue would have four slots 358. There would be many more invalidates than read data returns in a multi-processor system. Each entry or slot 357 in the invalidate queue includes an invalidate address 359, a type indicator, a status (valid) bit 360, and a next pointer 361 which points to the slot number of the next entry in chronological sequence of receipt. A tail pointer 362 is maintained for the queue

355, and a separate tail pointer 363 is maintained for the queue 356; when a new entry is incoming on the bus 345 from the system bus 11, it is loaded to one of the queues 355 or 356 depending upon its type (invalidate or read data), and into the slot 357 or 358 in this queue as identified by the tail pointer 362 or 363. Upon each such load operation, the tail pointer 362 or 363 is incremented, wrapping around to the beginning when it reaches the end. Entries are unloaded from the queues 355 and 356 and sent on to the transmitter 348 via bus 347, and the slot from which an entry is unloaded is defined by a head pointer 364. The head pointer 364 switches between the queues 355 and 356; there is only one head pointer. The entries in queues 355 and 356 must be forwarded to the CPU 10 in the same order as received from the system bus 11. The head pointer 364 is an input to selectors 365, 366 and 367 which select which one of the entries is output onto bus 347. A controller 368 containing the head pointer 364 and the tail pointer 362 and 363 sends a request on line 369 to the transmitter 348 whenever an entry is ready to send, and receives a response on line 370 indicating the entry has been accepted and sent on to the bus 20. At this time, the slot just sent is invalidated by line 371, and the head pointer 364 is moved to the next pointer value 361 in the slot just sent. The next pointer value may be the next slot in the same queue 355 or 356, or it may point to a slot in the other queue. Upon loading an entry in the queues 355 or 356, the value in next pointer 361 is not inserted until the following entry is loaded since it is not known until then whether this will be an invalidate or a return data entry.

Detailed Description Text (184):
Preferably, the data RAM control is a state machine which executes any of the following tasks, upon instruction from the arbiter: DAT.sub.-- DREAD (reads four quadwords of data-stream data from the back-up cache 15 and sends them to the memory management unit); DAT.sub.-- IREAD (reads four quadwords of instruction-stream data from the back-up cache 15 and sends them to the memory management unit, and the task may be cancelled midstream if the IREAD is aborted by the memory management unit); DAT.sub.-- WB (reads four quadwords of data from the back-up cache 15 and sends them to the write-back queue (63 in FIG. 4); DAT.sub.-- RM.sub.-- WRITE (performs a read-modify-write operation on a quadword in the back-up cache 15); DAT.sub.-- WRITE.sub.-- BMO (performs a full quadword write on the back-up cache); and DAT.sub.-- FILL (writes fill data into the back-up cache 15, and merges write data with the fill, if necessary). When the data RAM control 474 has finished executing a task, the data RAM control notifies the arbiter 471.

Detailed Description Text (186):
In a first step 481, the arbiter gives highest priority to performing a de-allocate caused by a previous task. When a transaction such as a read miss causes a cache block to be de-allocated, this de-allocate always takes place in step 482 as the next data RAM task. In step 483, transactions in the in queue 61 are given the next-highest priority. Fills and cache coherency requests both arrive in the in queue 61, and then in step 484, the fill or cache coherency transaction at the head of the in queue is performed.

Detailed Description Text (191):
Turning now to FIG. 19, there is shown a flow chart of the steps followed by the arbiter 471 in servicing the D-read latch 299, the I-read latch 300, and the write queue 60. In steps 501, 502, 503, the source given priority asserts the address of its memory command upon the internal address bus 288 of the cache controller (see FIG. 4). If the memory command accesses an internal processor register (IPR) or I/O or a write unlock, as tested in step 504, then the command is completed in step 505. (To simplify implementation, the test in step 504 can be done concurrently with step 506 so that the fill CAM is always addressed and a hit always causes execution of a command other than a WRITE UNLOCK to stall.) If, however, the memory command accesses memory space, then in step 506, processing of the task is halted if there is a hit in the fill CAM. In this case, the memory space access conflicts with an outstanding fill or an outstanding READ LOCK. If, however, there is not a conflict with an outstanding fill or READ LOCK, then in step 507, execution branches depending on whether the cache is in the above-described error transition mode or whether the memory access is requested by an ownership command. If so, then in step 508, the tag RAMs are accessed to determine whether there is a cache hit in an owned block. If not, then if the cache is in the error transition mode, as tested in step 509, the back-up cache is bypassed and the read or write is sent directly to system

memory (12 in FIG. 1) in step 510. If, however, in step 509, the cache was not operating in the error transition mode, then in step 511, an ownership read is sent to memory, and in step 512, the fill CAM is set to record that the refill is in progress. Moreover, if the addressed block in the cache is owned, as tested in step 513, then in step 514, the cache block is de-allocated and written back to memory in the next task. In other words, in step 514, a flag is set which is inspected by the arbitrator in step 481 of FIG. 18 to determine whether a need to de-allocate was caused by the previous task.

Detailed Description Text (192):
If in step 508 there was a cache hit in an owned block of the back-up cache, then in step 515, execution branches depending on whether the cache is in the error transition mode. If so, then in step 516, an ownership transaction is sent to memory, and the memory block is de-allocated and written back to memory in the next task. From step 515 or 516, execution continues in step 517 to complete the command.

Detailed Description Text (193):
If in step 507 it was found that the cache was neither in the error transition mode nor the command was an ownership command, then in step 518, execution branches depending on whether there was a cache hit. If so, then the command is completed in step 517. If not, then execution branches to step 519, where a refill of the cache block is begun by sending a data read or instruction read to memory. The fact that the refill is in progress is recorded in the fill CAM in step 512, and if the address block in the cache is owned, as tested in step 513, then in step 514, the address block is de-allocated and written back to memory in the next task.

Other Reference Publication (1):
Archibald et al., "Cache Coherence Protocols: Evaluation Using a Multi-Processor Simulation Model," ACM Transactions on Computer Systems, No. 4, Nov. 1986, New York, N.Y., U.S.A., pp. 273-298.

CLAIMS:

1. A method of operating a first processor in a multi-processor digital computer system having said first processor, a second processor, and a system memory accessed by both of said first and second processors over a system bus operating in accordance with a block ownership cache coherency protocol, said first processor having a cache memory for storing blocks of data in association with memory addresses, said method comprising the steps of:

a) fetching data having a specified memory address for a data processing operation by searching said cache memory for said specified memory address, and when said specified memory address is not found in said cache memory, storing the specified memory address in a content addressable memory, and sending a fill data request including said specified memory address to the system memory;

b) before receipt of fill data from the system memory,

i) receiving a cache coherency request from said second processor in accordance with said block ownership cache coherency protocol, said cache coherency request including said specified memory address and requesting invalidation of a block of data having the specified memory address, and

ii) checking whether said specified memory address is stored in said content addressable memory, delaying execution of said cache coherency request until said fill data is received from said system memory; and

c) receiving said fill data from said system memory, and using said fill data for said data processing without retaining a validated block of said fill data in said cache memory.

3. The method as claimed in claim 1, wherein said cache coherency request is a read invalidate request requesting said first processor to relinquish any exclusive privilege to write to said block of data having said specified memory address, and

wherein said method includes clearing an indication of ownership associated with said block of said fill data in said cache memory so that a block of said fill data validated for writing is not retained in said cache memory, and writing said block of said fill data in said cache memory back to said system memory.

8. The method as claimed in claim 7, wherein said data processing operation includes the writing of data to said specified memory address, said fill data request includes a request for an exclusive privilege to write to said block of data having said specified memory address, and wherein said method further includes writing said block of said fill data in said cache memory back to said system memory.

10. A method of operating a first processor in a multi-processor digital computer system having said first processor, a second processor, and a system memory accessed by both of said first and second processors over a system bus operating in accordance with a block ownership cache coherency protocol, said first processor having a cache memory for storing blocks of data and a memory address associated with each of said blocks of data, said method comprising the steps of:

a) fetching data having a specified memory address for a data processing operation by said first processor by searching said cache memory for said specified memory address, and when said specified memory address is not found in said cache memory, storing the specified memory address in an entry of a content addressable memory having a plurality of entries, and sending a fill data request including said specified memory address to said system memory;

b) before receipt of fill data from the system memory,

i) receiving a cache coherency request from said second processor in accordance with said block ownership cache coherency protocol, said cache coherency request including said specified memory address, and

ii) addressing said content addressable memory with the specified memory address of said cache coherency request, and upon finding that the specified memory address of said cache coherency request is in said entry of said content addressable memory, setting in said content addressable memory an indication that said cache coherency request is pending for said specified memory address; and

c) receiving said fill data from said system memory, using said fill data for said data processing operation by said first processor, checking said entry of said content addressable memory for said indication that said cache coherency request is pending for said specified memory address, executing said cache coherency request.

12. The method as claimed in claim 10, wherein said data processing operation includes the writing of data to said specified memory address, said fill data request includes a request for an exclusive privilege to write to said specified memory address, said method includes setting in said entry of said content addressable memory an indication of said request for an exclusive privilege to write to said specified memory address, said cache coherency request is a read invalidate request requesting said first processor to relinquish any exclusive privilege to write to a block of data having said specified memory address, said step (b) (ii) further includes checking whether said entry indicates said request for an exclusive privilege to write to said specified memory address, and wherein the setting in said content addressable memory of an indication that said cache coherency request is pending for said specified memory address is performed upon finding that said entry indicates said request for an exclusive privilege to write to said specified memory address.

13. The method as claimed in claim 10, wherein said data processing operation includes the writing of data to said specified memory address, said fill data request includes a request for an exclusive privilege to write to said specified memory address, said method includes setting in said entry of said content addressable memory an indication of said request for an exclusive privilege to write to said specified memory address, said cache coherency request is an ownership-read invalidate request requesting said first processor to refrain from reading or writing to a block of data having said specified memory address, and wherein step

(c) further includes checking whether said entry indicates said request for an exclusive privilege to write to said specified memory address, and upon finding that said entry indicates said request for an exclusive privilege to write to said specified memory address, writing a block of fill data including data written in accordance with said data processing operation back to said system memory.

14. The method as claimed in claim 10, wherein said cache coherency request is a read invalidate request requesting said first processor to relinquish any exclusive privilege to write to a block of data having said specified memory address, said method includes storing said fill data in a cache block in said cache memory, and wherein the execution of said cache coherency request includes clearing an indication of ownership associated with said cache block in said cache memory storing said fill data, and writing said cache block of fill data back to said system memory.

16. A processor for a multi-processor computer system, said multi-processor computer system having a system bus for coupling processors to a system memory, said system bus operating in accordance with a block ownership cache coherency protocol, said processor comprising, in combination:

instruction decoding means for decoding computer program instructions to generate requests for reading data at specified read addresses;

instruction execution means connected to said instruction means for executing the computer program instructions decoded by said instruction decoding means to generate requests for writing data at specified write addresses;

a cache memory for storing blocks of data, and in association with each block of data, a memory address, an indication of whether each block is valid for providing data from said memory address in response to said requests for reading data, and an indication of whether each block is valid for receiving data from said requests for writing data to said memory address;

a content addressable memory including a plurality of entries and means for storing in each entry a fill address of a fill request to a system memory in said multi-processor system requesting fill data from said fill address in said shared memory, an indication of whether the fill address is associated with a request for validation for writing data to said fill address, an indication of whether a read invalidate request was received, before return of said fill data, from another processor in said multi-processor system requesting invalidation of any indication that a cache block having said fill address in said cache memory is valid for receiving write data of whether an ownership-read invalidate request was received, before return of said fill data, from another processor in said multi-processor system requesting invalidation of any indication that a cache block having said fill address is valid for providing read data;

means, responsive to a request for reading data from a specified read address, for addressing said cache memory with said read address, for reading data from said cache memory when said cache memory contains a cache block having said read address and indicated as valid for providing read data, and when said cache memory does not contain a cache block having said read address and indicated as valid for providing read data, for sending a fill request to said main memory including said read address and for storing said read address in said content addressable memory;

means, responsive to a request for writing data to a specified write address, for writing data to said cache memory when said cache memory contains a cache block having said write address and indicated as valid for receiving write data, and when said cache memory does not contain a cache block having said write address and indicated as valid for receiving write data, for sending a fill request to said system memory including said write address and a request for validation for a write operation, and for storing in said content addressable memory said write address together with an indication that the fill address is associated with a request for validation for a write operation;

means, responsive to receiving from another processor in said multi-processor system

a read invalidate request having a specified read invalidate address, for addressing said content addressable memory with said specified read invalidate address, and when a fill address matching said specified read invalidate address is found in said content addressable memory, for setting the indication of whether a read invalidate request was received from another processor in said multi-processor system before return of said fill data;

means, responsive to receiving from another processor in said multi-processor system an ownership-read invalidate request having a specified ownership-read invalidate address, for addressing said content addressable memory with said specified ownership-read invalidate address, and when a fill address matching said specified ownership-read invalidating address is found in said content addressable memory, for setting the indication of whether an ownership-read invalidate request was received from another processor in said multi-processor system before return of said fill data;

first means, responsive to return of said fill data for checking said indication in said content addressable memory of whether an ownership-read invalidate request was received before return of said fill data, and when an ownership-read invalidate request was received before return of said fill data, for invalidating an indication that a cache block having the fill address in said cache memory is valid for providing read data; and second means, responsive to return of said fill data, for checking the indication in said content addressable memory of whether a read invalidate request was received before return of said fill data, and when a read invalidate request was received before return of said fill data, for invalidating an indication that a cache block having the fill address in said cache memory is valid for receiving write data.

17. The processor as claimed in claim 16, wherein said means, responsive to receiving from another processor in said multi-processing system a read invalidate request, further includes means for checking whether the matching fill address is associated with an indication of a request for validation for a write operation, and wherein said means for setting the indication of whether a read invalidate request was received does not set said indication of whether a read invalidate request was received when the matching fill address is not associated with an indication of a request for validation for a write operation.

WEST

End of Result Set

Generate Collection Print

L19: Entry 1 of 1 File: USPT Apr 4, 1995

DOCUMENT-IDENTIFIER: US 5404483 A
TITLE: Processor and method for delaying the processing of cache coherency transactions during outstanding cache fills

Abstract Text (1):
A processor and method for delaying the processing of cache coherency transactions during outstanding cache fills in a multi-processor system using a shared memory. A first processor fetches data having a specified address by addressing a cache memory, and when the specified address is not in the cache, saving the specified address in a fill address memory, and sending a fill request to the shared memory. Before return of fill data, the first processor receives a cache coherency request including the specified address from a second processor requesting invalidation of an addressed block of data. The first processor responds by checking whether the fill address memory includes the specified address, and upon finding the specified address in the fill address memory, delaying execution of the cache coherency request until the fill data is returned, and when the fill data is returned, using the fill data without retaining a validated block of the fill data in the cache. In a preferred embodiment, the fill memory is a content-addressable memory including a plurality of entries, and each entry has a fill address, an ownership fill bit (OREAD), an ownership-read invalidate pending bit (OIP), and a read invalidate pending bit (RIP). The OIP or RIP bit is set when execution of a cache coherency request is delayed, and these bits are read upon completion of a fill to execute the delayed request.

US Patent No. (1):
5404483

Parent Case Text (2):
The present application is a continuation-in-part of Ser. No. 07/547,699, filed Jun. 29, 1990, entitled BUS PROTOCOL FOR HIGH-PERFORMANCE PROCESSOR, by Rebecca L. Stamm et al., now abandoned in favor of continuation application Ser. No. 08/034,581, filed Mar. 22, 1993, entitled PROCESSOR SYSTEM WITH WRITEBACK CACHE USING WRITEBACK AND NON WRITEBACK TRANSACTIONS STORED IN SEPARATE QUEUES, by Rebecca L. Stamm, et al., issued on May 31, 1994, as U.S. Pat. No. 5,317,720, and Ser. No. 07/547,597, filed Jun. 29, 1990, entitled ERROR TRANSITION MODE FOR MULTI-PROCESSOR SYSTEM, by Rebecca L. Stamm et al., issued on Oct. 13, 1992, as U.S. Pat. No. 5,155,843, incorporated herein by reference. The present application is related to Stamm et al., "Preventing Access to Locked Memory Block By Recording Lock in Content Addressable Memory with Outstanding Cache Fills," Ser. No. 07/902,122, filed Jun. 22, 1992, concurrently with the present application.

Brief Summary Text (1):
The present application is related to Stamm et al., "Preventing Access to Locked Memory Block By Recording Lock in Content Addressable Memory with Outstanding Cache Fills," Ser. No. 07/902,122, filed Jun. 22, 1992, concurrently with the present application.

Brief Summary Text (8):
One known cache coherency protocol for a multi-processor system employing writeback caches is based on the concept of block ownership; an addressed portion of memory the size of a cache block is either owned by the shared memory or it is owned by one

of the writeback caches. Only one of the processors, or the shared memory, may own the block of memory at any given time, and this ownership is indicated by an ownership bit for each block in the shared memory and in each of the caches. A processor may write to a block only when the processor owns the block. Therefore the ownership bits always identify a unique "valid" block in the system. Shared read-only access to a block is permitted only when the shared memory owns the block. To indicate whether a processor may read a block, each of the caches includes, for each block, a "valid" bit. When a processor desires to read a block that is not valid in its cache, it issues a read transaction to the shared memory, requesting the shared memory to fill its cache with valid data. When a processor desires to write to a block which it does not own, it issues an ownership-read transaction to the shared memory, requesting ownership as well as a fill. From the perspective of the other processors, these transactions are cache coherency transactions, which request any other processor having ownership to give up ownership and writeback the data of the requested block, and in the case of an ownership read transaction, further request the other processors to invalidate any copies of the requested block.

Brief Summary Text (9):
Typically the time for a cache coherency transaction to be transmitted over a system bus is much shorter than the time for fill data to be retrieved from the shared memory. Therefore system performance can be improved by use of a pended bus (i.e., a bus which permits more than one transaction to be pending on the bus at any given time). The use of such a "pended" bus, however, leads to a problem of data coherency where a processor may issue a read transaction to fill a cache block, but before receiving the fill data, the processor may receive a cache coherency transaction requesting the cache block to be disowned or invalidated. A conventional way of handling this problem is for the processor to immediately invalidate the cache block by clearing the "valid" bit, which effectively discards the fill data when it is received. But in this case the processor which first requested the data does not get to use it, and must reissue a read transaction to get the data.

Brief Summary Text (11):
In accordance with a basic aspect of the invention, there is provided a method of operating a first processor in a multi-processor system that has a second processor and a shared memory accessed by both of the first and second processors. The first processor has a cache memory for storing blocks of data and associated addresses. The first processor fetches data having a specified address by addressing the cache memory with the specified address, and when the specified address is not found in the cache memory, saving the specified address in a fill address memory, and sending a fill data request including the specified address to the shared memory. Before receipt of the requested fill data from the shared memory, the first processor receives a cache coherency request including the specified address from the second processor requesting invalidation of a block of data having the specified address. The first processor responds to the cache coherency request by checking whether the fill memory includes the specified address, and upon finding that the fill memory includes the specified address, delaying execution of the cache coherency request until the fill data is received by the first processor, and when the fill data is received by the first processor, using the fill data without retaining a validated block of the fill data in the cache.

Brief Summary Text (12):
In a preferred embodiment, the fill memory is a content-addressable memory (CAM) including a plurality of entries, and each of the entries has a fill address, and associated with the fill address, an ownership fill bit (OREAD), an ownership-read invalidate pending bit (OIP), and a read invalidate pending bit (RIP). The ownership fill bit (OREAD) is set when the first processor requests a fill with ownership of a block of fill data having the associated fill address. The ownership-read invalidate pending (OIP) bit is set when the first processor receives from the second processor a request for ownership of the block of data having the associated fill address. The read invalidate pending (RIP) bit is set when the first processor receives from the second processor a request to read the block of data having the associated fill address, and the ownership fill bit (OREAD) associated with the fill address has been set. Upon receipt of a block of fill data from the shared memory, the first processor loads the block of fill data into the cache memory and subsequently uses

the fill data for a data processing operation. Immediately after loading the block of fill data into the cache memory, however, the block of fill data is invalidated when the associated OIP bit has been set, and the block of fill data is disowned and written back to the shared memory when the associated OREAD bit has been set and the associated OIP or RIP bit has been set. Finally, the first processor clears the entry in the fill memory having the address of the block of fill data.

Drawing Description Text (3):
FIG. 1 is a block diagram of a multi-processor computer system incorporating the present invention;

Drawing Description Text (4):
FIG. 2 is a block diagram of a primary cache memory of the CPU of FIG. 1;

Drawing Description Text (6):
FIG. 4 is a block diagram of a writeback cache controller used in the processors of the computer system of FIG. 1;

Drawing Description Text (9):
FIG. 7 is a block diagram of the bus interface and arbiter unit of the computer system of FIG. 1;

Drawing Description Text (10):
FIG. 8 is a block diagram of the invalidate queue and the return queue in the bus interface and arbiter unit of FIG. 7;

Drawing Description Text (13):
FIG. 11 is a block diagram showing a format of data stored in an entry in the write queue of FIG. 10;

Drawing Description Text (19):
FIG. 17 is a block diagram of control logic in the back-up cache controller of FIG. 4;

Detailed Description Text (8):
Each additional CPU 28 can include its own CPU chip 10, cache 15 and interface unit 21, if these CPUs 28 are of the same design as the CPU 10. Alternatively, these other CPUs 28 may be of different construction but executing a compatible bus protocol to access the main system bus 11. These other CPUs 28 can access the memory 12, and so the blocks of data in the caches 14 or 15 can become obsolete. If a CPU 28 writes to a location in the memory 12 that happens to be duplicated in the cache 15 (or in the primary cache 14), then the data at this location in the cache 15 is no longer valid. For this reason, blocks of data in the caches 14 and 15 are "invalidated" as will be described, when there is a write to memory 12 from a source other than the CPU 10 (such as the other CPUs 28). The cache 14 operates on a "write-through" principle, whereas the cache 15 operates on a "write-back" principle. When the CPU 10 executes a write to a location which happens to be in the primary cache 14, the data is written to this cache 14 and also to the backup cache 15 (and sometimes also to the memory 12, depending upon conditions); this type of operation is "write-through". When the CPU 10 executes a write to a location which is in the backup cache 15, however, the write is not necessarily forwarded to the memory 12, but instead is written back to memory 12 only if another element in the system (such as a CPU 28) needs the data (i.e., tries to access this location in memory), or if the block in the cache is displaced (deallocated) from the cache 15.

Detailed Description Text (12):
The memory management unit 25 receives read requests from the instruction unit 22 (both instruction stream and data stream) and from the execution unit 23 (data stream only). The memory management unit 25 delivers memory read data to either the instruction unit 22 (64-bits wide) or the execution unit 23 (32-bits wide). The memory management unit 25 also receives write/store requests from the execution unit 23, as well as invalidates, primary cache 14 fills and return data from the cache controller unit 26. The memory management unit 25 arbitrates between these requesters, and queues requests which cannot currently be handled. Once a request is started, the memory management unit 25 performs address translation, mapping virtual

to physical addresses, using a translation buffer. This address translation takes one machine cycle, unless there is a miss in the translation buffer. In the case of a miss, the memory management unit 25 causes a page table entry to be read from page tables in memory and a translation buffer fill is performed to insert the address which missed. The memory management unit also performs all access checks to implement page protection.

Detailed Description Text (13):
The primary cache 14 referenced by the memory management unit 25 is a two-way set associative write-through cache with a block and fill size of 32-bytes. The primary cache state is maintained as a subset of the backup cache 15.

Detailed Description Text (15):
In response to a memory read request (other than a READ LOCK, as described below), the memory management unit 25 accesses the primary cache 14 for the read data. If the primary cache 14 determines that requested read data is not present, a "cache miss" or "read miss" condition occurs. In this event, the memory management unit 25 instructs the cache controller unit 26 to continue processing the read. The cache controller unit 26 first looks for the data in the backup cache 15 and fills the block in the primary cache 14 from the backup cache 15 if the data is present. If the data is not present in the backup cache 15, the cache controller unit 26 requests a cache fill on the CPU bus 20 from memory 12. When memory 12 returns the data, it is written to both the Backup cache 15 and to the primary cache 14. The cache controller unit 26 sends four quadwords of data to the memory management unit 25 using instruction-stream cache fill or data-stream cache fill commands. The four cache fill commands together are used to fill the entire primary cache 14 block corresponding to the hexaword (16-word) read address on bus 57. In the case of data-stream fills, one of the four cache fill commands will be qualified with a signal indicating that this quadword fill contains the requested data-stream data corresponding to the quadword address of the read. When this fill is encountered, it will be used to supply the requested read data to the memory management unit 25, instruction unit 22 and/or execution unit 23. If, however, the physical address corresponding to the cache fill command falls into I/O space, only one quadword fill is returned and the data is not cached in the primary cache 14. Only memory data is cached in the primary cache 14.

Detailed Description Text (19):
Cache coherency in the multi-processor system of FIG. 1 is based upon the concept of ownership; a hexaword (16-word) block of memory may be owned either by the system memory 12 or by a backup cache 15 in a CPU on the bus 11--in a multi-processor system. Only one of the caches, or system memory 12, may own the hexaword block at a given time, and this ownership is indicated by an ownership bit for each hexaword in both memory 12 and the backup cache 15 (1 for own, 0 for not-own) .

Detailed Description Text (20):
Shared read-only access to a block among the CPUs 10, 28 is permitted only when system memory 12 owns the block. A CPU may write to a block only when the CPU owns the block. These rules ensure that there is always a unique "valid" block of data in the system, identified by the ownership bits in the caches, and a CPU will always read data from the valid block and write data to the valid block.

Detailed Description Text (21):
The multi-processor system follows certain protocols which ensure rapid access to the valid data of an addressed block. Each back-up cache 15 maintains two bits associated with each cache block. These two bits are called VALID and OWNED, and they determine the state of the cache block as shown in TABLE A.

Detailed Description Text (22):
If the VALID bit of a cache block is not set, then the cache block is invalid. If an invalid cache block is accessed by its CPU, then the cache block is refilled with data from the current owner of the cache block. If the access is for a data read operation and the current owner is the system memory 12, then the refill data is obtained from the system memory 12, and the system memory 12 will retain ownership of the cache block. If the access is for a data read operation and the current owner is another cache, then the valid data is written from that other cache back to the

system memory 12, the valid data is also refilled in the cache block of the cache of the accessing CPU, and the system memory 12 obtains and retains ownership of the cache block. If the access is for a write operation and the current owner is the system memory 12, then the valid data is obtained from the system memory 12, and the accessing cache obtains ownership of the cache block. If the access is for a write operation and the current owner is another cache, then the valid data is written from that other cache back to the system memory 12, the cache block of that other cache is invalidated, the valid data is also refilled in the cache block of the cache of the accessing CPU, and the cache of the accessing CPU obtains ownership of the cache block.

Detailed Description Text (23):
If the VALID bit of the cache block of the accessing CPU is set but the corresponding OWNED bit is not set, then the system memory 12 is the owner of the cache block. The accessing CPU can read valid data from this valid un-owned cache block. The accessing CPU, however, can write data to this valid un-owned cache block only after obtaining ownership of the cache block from the system memory 12 and invalidating any copies in other caches.

Detailed Description Text (24):
If both the VALID and OWNED bits of the cache block of the accessing CPU are set, then the accessing CPU is the owner of the cache block, and the accessing CPU is free to read or write to the cache block.

Detailed Description Text (25):
A cache can also "disown" ownership of a cache block, for example, when a cache block currently owned by the cache is written back to memory to free-up space in the cache for another cache block.

Detailed Description Text (27):
The instruction read command IREAD requests instructions from an addressed cache block. The DREAD command requests data from an addressed cache block. When intercepted by another CPU, these commands cause no change in the state of the cache of another CPU unless the accessed cache block is owned by another CPU. In this case, the other CPU relinquishes ownership by writing the data in the cache block of its cache back to system memory 12 and setting the cache block of its own cache to a state of "valid-unowned." This kind of writeback-invalidate operation is known as a "Rinval" operation.

Detailed Description Text (28):
The command OREAD requests ownership as well as data from the addressed cache block. The command WRITE transmits data to the system memory 12. If another CPU intercepts either of these commands and has the addressed cache block in its cache, then it invalidates the addressed block in its cache. Moreover, if the other CPU owned the addressed cache block, it gives up ownership and writes back data from the addressed cache block in its cache to the system memory 12. This kind of writeback-invalidate operation is known as an "Oinval" operation.

Detailed Description Text (29):
To avoid stalls and possible deadlocks, the above cache coherency protocols are implemented in such a manner as to pass cache block ownership from one CPU to another as quickly as possible. In this regard, cache block ownership is different from a memory lock that typically requires execution of respective program instructions for setting and clearing the lock. The hardware of the preferred embodiment of FIG. 1, for example, does not have such memory locking facilities, which could be implemented by storing additional "lock bits" in association with the cache blocks in each of the caches and the system memory 12.

Detailed Description Text (30):
The preferred embodiment of FIG. 1, however, does have one instance where one CPU will not immediately relinquish ownership to another CPU. The preferred embodiment executes (VAX) instructions, including certain "interlocked" instructions that are guaranteed to perform atomic operations upon memory in a multi-processing environment. An example is an "add aligned word interlocked" instruction (ADAWI) which adds a first operand to a second operand and sets the second operand to the

sum. The destination operand has an access type of "modify" raising the possibility that one CPU might obtain ownership of a cache block between the time that the second operand is read from memory and the time that the second operand is modified and written back to memory, leading to a result in memory which might not appear consistent under certain program sequences. Computers which execute (VAX) instructions in a multi-processing environment typically prevent such an interruption of memory access by using the execution unit to request fetching of the second operand and to request a memory "read lock" when fetching the second operand from memory, and to request a memory "write unlock" when putting the result back to memory.

Detailed Description Text (31):
In the CPU 10 in FIG. 1, the execution unit 23 transmits to the memory management unit 25 a memory fetch and read lock request for fetching from memory an operand to be modified by an interlocked (VAX) instruction. In response to the read lock request, the memory management unit places a memory lock on the memory location to be modified. This memory lock remains until the execution unit sends the result to the locked memory location together with a memory unlock request. Since the execution unit generates paired of read lock/write unlock requests during execution of a single interlocked instruction, only one cache block is locked by this mechanism at any given time. Moreover, when the execution unit generates a read lock request, it will be followed, without interruption, by a corresponding write unlock request.

Detailed Description Text (33):
Upon receipt of a READ LOCK command, the cache controller obtains ownership of the cache block to be locked, if the cache block is not already owned, before transmitting the referenced data back to the memory management unit 25. Ownership of this interlocked cache block is retained at least until cache controller 26 writes the modified value back into the interlocked cache block upon receipt of a corresponding WRITE UNLOCK command from the memory management unit. Write-back of the block to the system memory 12 is prevented from the time that the cache controller receives the READ LOCK command to the time that the cache controller executes the WRITE UNLOCK command. Moreover, in the preferred system of FIG. 1, once a READ LOCK command has been passed to the cache controller, the cache controller will not process any subsequent data stream read references until the corresponding WRITE UNLOCK command has been executed.

Detailed Description Text (35):
The cache controller responds to an IREAD, DREAD, and READ MODIFY command in a similar fashion by accessing the back-up cache 15, and detecting a "cache hit" if the cache tag matches the requested cache block address and the valid bit of the indexed cache block is set. The back-up cache is accessed in a similar fashion for the READ LOCK command, but a cache bit also requires the ownership bit of the indexed cache block to be set. IREAD and DREAD misses result in IREAD and DREAD commands on the CPU bus 20 and the system bus 11. READ MODIFY, READ.sub.-- LOCK, and WRITE misses result in OREAD commands the CPU bus 20 and the system bus 11.

Detailed Description Text (37):
Turning now to FIG. 2, the primary cache 14 is a two-way set-associative, read allocate, no-write allocate, write-through, physical address cache of instruction stream and data stream data. The primary cache 14 has a one-cycle access and a one-cycle repetition rate for both reads and writes. The primary cache 14 includes an 8Kbyte data memory array 268 which stores 256-hexaword blocks, and stores 256 tags in tag stores 269 and 270. The data memory array 268 is configured as two blocks 271 and 272 of 128 rows. Each block is 256-bits wide so it contains one hexaword of data (four quadwords or 32-bytes); there are four quadword subblocks per block with a valid bit associated with each subblock. A tag is twenty bits wide, corresponding to bits <31:12> of the physical address on bus 243.

Detailed Description Text (38):
Turning now to FIG. 3, the organization of data in the primary cache 14 is shown in more detail. Each index (an index being a row of the memory array 268) contains an allocation pointer A, and contains two blocks where each block consists of a 20-bit tag, 1-bit tag parity TP, four valid bits VB (one for each quadword), 256-bits of

data, and 32-bits of data parity.

Detailed Description Text (39):
Returning now to FIG. 2, a row decoder 273 receives bits <5:11> of the primary cache 14 input address from the bus 243 and selects 1-of-128 indexes (rows) 274 to output on column lines of the memory array, and column decoders 275 and 276 select 1-of-4 colums based on bits <3:4> of the address. So, in each cycle, the primary cache 14 selects two quadword locations from the hexaword outputs from the array, and the selected quadwords are available on input/output lines 277 and 278. The two 20-bit tags from tag stores 269 and 270 are simultaneously output on lines 279 and 280 for the selected index and are compared to bits <31:12> of the address on bus 243 by tag compare circuits 281 and 282. The valid bits are also read out and checked; if zero for the addressed block, a miss is signaled. If either tag generates a match, and the valid bit is set, a hit is signalled on line 283, and the selected quadword is output on bus 246. A primary cache 14 miss results in a quadword fill; a memory read is generated, resulting in a quadword being written to the block 271 or 272 via bus 246 and bus 277 or 278. At the same time data is being written to the data memory array, the address is being written to the tag store 269 or 270 via lines 279 or 280. When an invalidate is sent by the cache controller unit 26, upon the occurrence of a write to backup cache 15 or system memory 12, valid bits are reset for the index.

Detailed Description Text (42):
A primary cache 14 fill operation is initiated by an instruction stream or data stream cache fill reference. A fill is a specialized form of a write operation, in which fill address bits <31:12> are written into the tag field of the selected bank. If a cache fill sequence to the same hexaword address is in progress when the Inval is executed, then any further cache fills are inhibited from loading data or validating data for this cache block.

Detailed Description Text (44):
Both the tags and data for the backup cache 15 are stored in off-chip RAMs, with the size and access time selected as needed for the system requirements. The backup cache 15 may be of a size of from 128K to 2Mbytes, for example. With an access time of 28 nsec, the cache can be referenced in two machine cycles, assuming 14 nsec machine cycle for the CPU 10. The cache controller unit 26 packs sequential writes to the same quadword in order to minimize write accesses to the backup cache. Multiple write commands from the memory management unit 25 are held in an eight-entry write queue (60 in FIG. 4) in order to facilitate this packing, as further described below.

Detailed Description Text (49):
For a write request, write data enters the cache controller unit 26 from the data bus 58 into the write queue 60 while the write address enters from the physical address bus 57; if there is a cache hit, the data is written into the data RAMs of the backup cache 15 via bus 289 using the address on bus 288, via bus 19. When a writeback of the block occurs, data is read out of the data RAMs via buses 19 and 289, transferred to the writeback queue 63 via interface 303 and buses 291 and 292, then driven out onto the CPU bus 20. A read request enters from the physical address bus 57 and the latches 299 or 300 and is applied via internal address bus 288 to the backup cache 15 via bus 19, and if a hit occurs the resulting data is sent via bus 19 and bus 289 to the data latch 304 in the output latch 296, from which it is sent to the memory management unit 25 via data bus 58. When read data returns from system memory 12, it enters the cache controller unit 26 through the input queue 61 and is driven onto bus 292 and then through the interface 303 onto the internal data bus 289 and into the data RAMs of the backup cache 15, as well as to the memory management unit 25 via output latch 296 and bus 58 as before.

Detailed Description Text (50):
If a read or write incoming to the cache controller unit 26 from the memory management unit 25 does not result in a backup cache 15 hit, the miss address is loaded into the fill CAM 302, which holds addresses of outstanding read and write misses; the address is also driven through the interface 303 to the non-writeback queue 62 via bus 291; it enters the queue 62 to await being driven onto the CPU bus 20 in its turn. Many cycles later, the data returns on the CPU bus 20 (after

accessing the system memory 12) and enters the input queue 61. The CPU 10 will have started executing stall cycles after the backup cache 15 miss, in the various pipelines. Accompanying the returning data is a control bit on the control bus in the CPU bus 20 which says which one of the two address entries in the fill CAM 302 is to be driven out onto the bus 288 to be used for writing the data RAMs and tag RAMs of the backup cache 15.

Detailed Description Text (51):
When a cache coherency transaction appears on the CPU bus 20, an address comes in through the input queue 61 and is driven via bus 290 and interface 303 to the bus 288, from which it is applied to the tag RAMs of the backup cache 15 via bus 19. If it hits, the valid bit is cleared, and the address is sent out through the address latch 305 in the output latch 296 to the memory management unit 25 for a primary cache 14 invalidate (where it may or may not hit, depending upon which blocks of backup cache 15 data are in the primary cache 14). If necessary, the valid and/or owned bit is cleared in the backup cache 15 entry. Only address bits <31:5> are used for invalidates, since the invalidate is always to a hexaword.

Detailed Description Text (53):
A five-bit command bus 262 from the memory management unit 25 is applied to a controller 306 to define the internal bus activities of the cache controller unit 26. This command bus indicates whether each memory request is one of eight types: instruction stream read, data stream read, data stream read with modify, interlocked data stream read, normal write, write which releases lock, or read or write of an internal or external processor register. These commands affect the instruction or data read latches 299 and 300, or the write packer 301 and the write queue 60. Similarly, a command bus 262 goes back to the memory management unit 25, indicating that the data being transmitted during the cycle is a data stream cache fill, an instruction stream cache fill, an invalidate of a hexaword block in the primary cache 14, or a NOP. These command fields also accompany the data in the write queue, for example.

Detailed Description Text (56):
In FIG. 5, a timing diagram of the operation of the bus 20 during three cycles is shown. These three cycles are a null cycle-0 followed by a write sequence; the write address is driven out in cycle-1, followed by the write data in cycle-2. Trace (a) shows the data or address on the 64-bit data/address bus. Traces (b) to (e) show the arbitration sequence. In cycle-0 the CPU 10 asserts a request to do a write by a request line being driven low from P2 to P4 of this cycle, seen in trace (b). As shown in trace (d), the arbiter in the bus interface 21 asserts a CPU-grant signal beginning at P2 of cycle-0, and this line is held down (asserted) because the CPU 10 asserts the CPU-hold line as seen in trace (c). The hold signal guarantees that the CPU 10 will retain control of the bus, even if another node such as an I/O 13a or 13b asserts a request. The hold signal is used for multiple-cycle transfers, where the node must keep control of the bus for consecutive cycles. After the CPU releases the hold line at the end of P4 of cycle-1, the arbiter in the interface unit 21 can release the grant line to the CPU in cycle-2. The acknowledge line is asserted by the bus interface 21 to the CPU 10 in the cycle after it has received with no parity errors the write address which was driven by the CPU in cycle-1. Not shown in FIG. 5 is another acknowledge which would be asserted by the bus interface 21 in cycle-3 if the write data of cycle-2 is received without parity error. The Ack must be asserted if no parity error is detected in the cycle following data being driven.

Detailed Description Text (59):
The writeback-only or WB-only line 20k, when asserted by the arbiter 325, means that the node it is directed to (e.g., the CPU 10) will only issue write-disown commands, including write disowns due to write-unlocks when the cache is off. Otherwise, the CPU 10 will not issue any new requests. During the cycle in which the WB-only line 20k is asserted to the CPU 10, the system must be prepared to accept one more non-writeback command from the CPU 10. Starting with the cycle following the assertion of WB-only, the CPU 10 will issue only writeback commands. The separate writeback and non-writeback queues 63 and 62 in the cache controller unit 26 of FIG. 4 allow the queued transactions to be separated, so when the WB-only line 20k is asserted the writeback queue 62 can be emptied as needed so that the other nodes of the system continue to have updated data available in system memory 12.

Detailed Description Text (60):
When any node asserts its suppress line 20j, no transactions other than writebacks or fills must be driven onto the bus 20, starting the following cycle. For example, when the CPU 10 asserts its suppress line 20j, the arbiter 325 can accomplish this by asserting WB-only to both I/O 13a and I/O 13b, so these nodes do not request the bus except for fills and writebacks. Thus, assertion of suppress by the CPU 10 causes the arbiter 325 to assert WB-only to the other two nodes 13a and 13b. Or, assertion of suppress by I/O 13a will cause the arbiter 325 to assert WB-only to CPU 10 and I/O 13b. The hold line 20h overrides the suppress function.

Detailed Description Text (67):
The Bad Write Data command appearing on the bus 20b, as listed in Table C, functions to allow the CPU 10 to identify one bad quadword of write data when a hexaword writeback is being executed. The cache controller unit 26 tests the data being read out of the backup cache 15 on its way to the bus 20 via writeback queue 62. If a quadword of the hexaword shows bad parity in this test, then this quadword is sent by the cache controller unit 26 onto the bus 20 with a Bad Write Data command on the bus 20b, in which case the memory 12 will receive three good quadwords and one bad in the hexaword write. Otherwise, since the write block is a hexaword, the entire hexaword would be invalidated in memory 12 and thus unavailable to other CPUs. Of course, error recovery algorithms must be executed by the operating system to see if the bad quadword sent with the Bad Write Data command will be catastrophic or can be worked around.

Detailed Description Text (70):
The backup cache 15 for the CPU 10 is a "write-back" cache, so there are times when the backup cache 15 contains the only valid copy of a certain block of data, in the entire multi-processor system of FIG. 1. The backup cache 15 (both tag store and data store) is protected by ECC. Check bits are stored when data is written to the cache 15 data RAM or written to the tag RAM, then these bits are checked against the data when the cache 15 is read, using ECC check circuits 330 and 331 of FIG. 4. When an error is detected by these ECC check circuits, an Error Transition Mode is entered by the C-box controller 306; the backup cache 15 can't be merely invalidated, since other system nodes 28 may need data owned by the backup cache 15. In this error transition mode, the data is preserved in the backup cache 15 as much as possible for diagnostics, but operation continues; the object is to move the data for which this backup cache 15 has the only copy in the system, back out to system memory 12, as quickly as possible, but yet without unnecessarily degrading performance. For blocks (hexawords) not owned by the backup cache 15, references from the memory management unit 25 received by the cache controller unit 26 are sent to system memory 12 instead of being executed in the backup cache 15, even if there is a cache hit. For blocks owned by the backup cache 15, a write operation by the CPU 10 which hits in the backup cache 15 causes the block to be written back from backup cache 15 to system memory 12, and the write operation is also forwarded to system memory 12 rather than writing to the backup cache 15; only the ownership bits are changed in the backup cache 15 for this block. A read hit to a valid-owned block is executed by the backup cache 15. No cache fill operations are started after the error transition mode is entered. Cache coherency transactions from the system bus 20 are executed normally, but this does not change the data or tags in the backup cache 15, merely the valid and owned bits. In this manner, the system continues operation, yet the data in the backup cache 15 is preserved as best it can be, for later diagnostics.

Detailed Description Text (71):
Thus, when the cache controller unit 26 detects uncorrectable errors using the ECC circuits 330 and 331, it enters into Error Transition Mode (ETM). The goals of the cache controller unit 26 operation during ETM are the following: (1) preserve the state of the cache 15 as much as possible for diagnostic software; (2) honor memory management unit 25 references which hit owned blocks in the backup cache 15 since this is the only source of data in the system; (3) respond to cache coherency requests received from the bus 20 normally.

Detailed Description Text (72):
Once the cache controller unit 26 enters Error Transition Mode, it remains in ETM

until software explicitly disables or enables the cache 15. To ensure cache coherency, the cache 15 must be completely flushed of valid blocks before it is re-enabled because some data can become stale while the cache is in ETM.

Detailed Description Text (73):
Table D describes how the backup cache 15 behaves while it is in ETM. Any reads or writes which do not hit valid-owned during ETM are sent to system memory 12: read data is retrieved from system memory 12, and writes are written to memory 12, bypassing the cache 15 entirely. The cache 15 supplies data for Ireads and Dreads which hit valid-owned; this is normal cache behavior. If a write hits a valid-owned block in the backup cache 15, the block is written back to memory 12 and the write is also sent to system memory 12. The write leaves the cache controller unit 26 through the non-writeback queue 62, enforcing write ordering with previous writes which may have missed in the backup cache 15. If a Read-Lock hits valid-owned in the cache 15, a writeback of the block is forced and the Read-Lock is sent to system memory 12 (as an Owned-Read on the bus 20). This behavior enforces write ordering between previous writes which may have missed in the cache and the Write-Unlock which will follow the Read-Lock.

Detailed Description Text (74):
The write ordering problem alluded to is as follows: Suppose the cache 15 is in ETM. Also suppose that under ETM, writes which hit owned in the cache 15 are written to the cache while writes which miss are sent to system memory 12. Write A misses in the cache 15 and is sent to the non-writeback queue 62, on its way to system memory 12. Write B hits owned in the cache 15 and is written to the cache. A cache coherency request arrives for block B and that block is placed in the writeback queue 63. If Write A has not yet reached the bus 20, Writeback B can pass it since the writeback queue has priority over the non-writeback queue. If that happens, the system sees write B while it is still reading old data in block A, because write A has not yet reached memory. For this reasons, as noted below, all writes (except for a write unlock), are placed in the non-writeback queue during ETM.

Detailed Description Text (80):
Referring to FIG. 8, the response queue 346 employs separate queues 355 and 356 for the invalidates and for return data, respectively. The invalidate queue 355 may have, for example, twelve entries or slots 357 as seen in FIG. 9, whereas the return data queue would have four slots 358. There would be many more invalidates than read data returns in a multi-processor system. Each entry or slot 357 in the invalidate queue includes an invalidate address 359, a type indicator, a status (valid) bit 360, and a next pointer 361 which points to the slot number of the next entry in chronological sequence of receipt. A tail pointer 362 is maintained for the queue 355, and a separate tail pointer 363 is maintained for the queue 356; when a new entry is incoming on the bus 345 from the system bus 11, it is loaded to one of the queues 355 or 356 depending upon its type (invalidate or read data), and into the slot 357 or 358 in this queue as identified by the tail pointer 362 or 363. Upon each such load operation, the tail pointer 362 or 363 is incremented, wrapping around to the beginning when it reaches the end. Entries are unloaded from the queues 355 and 356 and sent on to the transmitter 348 via bus 347, and the slot from which an entry is unloaded is defined by a head pointer 364. The head pointer 364 switches between the queues 355 and 356; there is only one head pointer. The entries in queues 355 and 356 must be forwarded to the CPU 10 in the same order as received from the system bus 11. The head pointer 364 is an input to selectors 365, 366 and 367 which select which one of the entries is output onto bus 347. A controller 368 containing the head pointer 364 and the tail pointer 362 and 363 sends a request on line 369 to the transmitter 348 whenever an entry is ready to send, and receives a response on line 370 indicating the entry has been accepted and sent on to the bus 20. At this time, the slot just sent is invalidated by line 371, and the head pointer 364 is moved to the next pointer value 361 in the slot just sent. The next pointer value may be the next slot in the same queue 355 or 356, or it may point to a slot in the other queue. Upon loading an entry in the queues 355 or 356, the value in next pointer 361 is not inserted until the following entry is loaded since it is not known until then whether this will be an invalidate or a return data entry.

Detailed Description Text (81):
The interface chip 21 provides the memory interface for CPU 10 by handling CPU

memory and I/O requests on the system bus 11. On a memory Read or Write miss in the backup cache 15, the interface 21 sends a Read on system bus 11 and receives a cache fill operation to acquire the block from system memory 12. The interface chip 21 monitors memory Read and Write traffic generated by other nodes on the system bus 11 such as CPUs 28 to ensure that the CPU 10 caches 14 and 15 remain consistent with main system memory 12. If a Read or Write by another node hits the cache 15, then a Writeback or Invalidate is performed by the CPU 10 chip as previously discussed. The interface chip 21 also handles interrupt transactions to and from the CPU.

Detailed Description Text (82):
The system bus 11 includes a suppress signal as discussed above with respect to the CPU bus 20 (i.e., line 20j), and this is used to control the initiation of new system bus 11 transactions. Assertion of suppress on the system bus 11 blocks all bus commander requests, thus suppressing the initiation of new system bus 11 transactions. This bus 11 suppress signal may be asserted by any node on bus 11 at the start of each bus 11 cycle to control arbitration for the cycle after the next system bus 11 cycle. The interface chip 21 uses this suppress signal to inhibit transactions (except Writeback and Read Response) on the system bus 11 when its invalidate queue 355 is near full in order to prevent an invalidate queue 355 overflow.

Detailed Description Text (88):
For Read Miss and Fill operations, when a read misses in the CPU 10 CPU, the request goes across the bus 20 to the interface chip 21. When the memory interface returns the data, the CPU 10 cache controller unit 26 puts the fill into the in-queue 61. Since the block size is 32-bytes and the bus 20 is 8-bytes wide, one hexaword read transaction on the bus 20 results from the read request. As fill data returns, the cache controller unit 26 keeps track of how many quadwords have been received with a two-bit counter in the fill CAM 302. If two read misses are outstanding, fills from the two misses may return interleaved, so each entry in the fill CAM 302 has a separate counter. When the last quadword of a read miss arrives, the new tag is written and the valid bit is set in the cache 15. The owned bit is set if the fill was for an Ownership Read.

Detailed Description Text (91):
For deallocates due to CPU Reads and Writes, when any CPU 10 tag lookup for a read or a write results in a miss, the cache block is deallocated to allow the fill data to take its place. If the block is not valid, no action is taken for the deallocate. If the black is valid but not owned, the block is invalidated. If the block is valid and owned, the block is sent to the interface chip 21 on the bus 20 and written back to system memory 12 and invalidated in the tag store. The Hexaword Disown Write command is used to write the data back. If a writeback is necessary, it is done immediately after the read or write miss occurs. The miss and the deallocate are contiguous events for the cache controller and are not interrupted for any other transaction.

Detailed Description Text (98):
The CPU 10, however, uses pended busses 11 and 20, and invalidates travel along the same path as the return data. It is necessary to retain strict order of transmission, so that invalidates and return data words must be sent to the CPU 10 for processing in exactly the same order that they entered the queue 346 from the system bus 11. This goal could be accomplished by simply having one unified queue, large enough to handle either invalidates or return data words, but this would unduly increase the chip size for the interface chip 21. Specifically, in practice, one unified queue means that each slot would have to be large enough to accommodate the return data, since that word is the larger of the two. In fact, the return data word and its associated control bits are more than twice as large as the invalidate address and its control bits. The invalidate portion of the queue will also have to be around twice the size of the return data portion. Thus, around 2/3 of the queue would be only half utilized, or 1/3 of the queue being wasted.

Detailed Description Text (102):
The approach of FIGS. 8 and 9 has several advantages over the use of a single queue, without greatly increasing the complexity of the design. The advantages all pertain to providing the necessary performance, while reducing the chip size. The specific

main advantages are: (1) The same performance obtained with a large, unified queue can be realized with far less space using the split queue method; (2) Each queue can be earmarked for a specific type of data, and there can be no encroaching of one data type into the other. As such, the two types of queues (invalidate and return data) can be tuned to their optimum size. For example, the invalidate queue might be seven (small) slots while the read data queue might be five or six (large) slots. This would provide a smooth read command overlap, while allowing invalidates to be processed without unduly suppressing the system bus 11; (3) The read data queue 356 can be increased to accommodate two outstanding reads without worrying about the size of the invalidate queue, which can remain the same size, based upon its own needs.

Detailed Description Text (105):
As introduced above, the fill CAM 302 in FIG. 4 holds addresses of outstanding misses to the back-up cache 15. By accessing the fill CAM before accessing the back-up cache 15, further access to the missed cache block for another memory management unit command or a cache coherency transaction is stalled until the fill is completed. When the cache is off or in ETM, however, writes are not checked for block conflict, but are sent immediately to memory.

Detailed Description Text (107):
A miss to a cache block in the back-up cache 15 is outstanding until the fill data has been received from the system memory 12. When a read transaction is issued to the system memory 12 to request the fill data, the fill CAM entry is validated by setting the valid bit, the address field is loaded, and the appropriate status bits RDLK, IREAD, OREAD, WRITE, and TO.sub.-- MBOX are set depending on the particular command, from the memory management unit, that required the access to the back-up cache 15. RIP, OIP, RDLK.sub.-- FL.sub.-- DONE, and REQ.sub.-- FILL.sub.-- DONE are cleared. If the cache is off, in ETM, or the miss is for an I/O reference, DNF is set; otherwise, it is cleared. COUNT is set to zero if four fill quadwords are expected; it is set to 3 if only one quadword is expected.

Detailed Description Text (109):
When the CPU 10 receives a cache coherency transaction from the CPU bus 20, the cache block address of the transaction is compared to the addresses in the fill CAM 302. If there is a match and the matching entry is valid, then the transaction is addressed to a cache block which has an outstanding fill request. The transaction is handled as shown in TABLE G. If the transaction is OREAD or WRITE (i.e., an ownership invalidating transaction), the fill CAM status bit OIP (OREAD invalidate pending) in the matching entry is set, and an invalidate is sent immediately to the primary cache 14. If the transaction is DREAD or IREAD (i.e., a read invalidating transaction) and the OREAD bit in the matching entry is set, then the fill CAM status bit RIP in the matching entry is set.

Detailed Description Text (110):
As the fills are received, the fill data is forwarded to the memory management unit 25 for use by the CPU 10. When all of the fills for the outstanding miss are received by the cache controller 26, and DNF is not set, then the action taken immediately after the fill is complete is dependent on the state of the RIP and OIP status bits of the fill CAM entry corresponding to the outstanding miss, as specified in TABLE H. In particular, if OIP is set and DNF is not set, then an Oinval operation is performed; the just-filled cache block is written back to memory if the refilled cache block is owned by the cache, and the just-filled cache block is invalidated by clearing both VALID and OWNED in the cache block. If RIP is set and DNF is not set, then an Rinval operation is performed; the just-filled cache block is written back to memory, and the just-filled cache block is set to a valid-unowned state by clearing OWNED in the cache block.

Detailed Description Text (111):
There are several error cases where RIP or OIP may be set, indicating the need for a cache coherence transaction, but the cache controller 26 will not execute the transaction. The fill sequence, for example, may fail by ending in RDE (Read Error) or by not refilling the cache within a predetermined duration of time (a fill timeout error). If the fill was meant for the primary cache 14 and ends in an error, the primary cache invalidates itself. Another error case, further described below,

occurs when a READ LOCK sequence does not conclude with a corresponding WRITE UNLOCK, but instead concludes with a write-one-to-clear to the RDLK bit to an error status register (CEFSTS; 308 in FIG. 4).

Detailed Description Text (114):
When the cache controller 26 receives a READ LOCK command from the memory management unit 25, further access to the cache block specified by the READ LOCK command must be stalled until the corresponding WRITE UNLOCK command is received and executed by the cache controller 26, as was introduced above. One way to perform this function would be to store the address of the outstanding read lock in a separate register, and to check the address in this register of any new memory access command from the memory management unit 25 or cache coherency transaction from the CPU bus 20; if the address matched, that command or transaction would be stalled until the corresponding WRITE LOCK command would be executed. In the cache controller 26 of FIG. 4, however, the fill CAM 302 is used to obtain the same result. The primary purpose of the fill CAM 302 is to hold the addresses and other information related to memory access commands which have missed in the back-up cache so that further accesses to those cache blocks can be prevented until the cache fills are returned from memory. But the fill CAM 302 is also used to hold outstanding READ LOCK information, so that access to a locked cache block is also prevented until the corresponding WRITE UNLOCK is executed.

Detailed Description Text (115):
In a preferred arrangement, when the cache controller 26 receives a READ LOCK command from the memory management unit 25, the cache controller places the block address specified by the READ LOCK command into an entry of the fill CAM 302, regardless of whether or not the block address hits in the back-up cache 15. At the same time, the following control bits are set in that fill CAM entry: RDLK (to indicate that a READ LOCK is in progress); OREAD (to indicate that the READ LOCK is an Ownership-Read type of transaction); TO.sub.-- MBOX (if the returning fill data is to be sent to the memory management unit 25); and VALID (to indicate that the entry is currently valid).

Detailed Description Text (116):
While the READ LOCK is in progress (i.e., recorded in the fill CAM) and before the corresponding WRITE UNLOCK is executed, the cache controller 26 may receive a cache coherency transaction from the CPU bus. Such a transaction may eventually result in either an invalidate of a cache block (Rinval) or a deallocate of an owned cache block (Oinval). Such a result must be prevented so long as the READ LOCK is in progress upon the cache block referenced by the cache coherency transaction. When such a transaction is received, its address is compared to any valid address in the fill CAM 302, including any READ LOCK address in the fill CAM. If the comparison indicates a match, then either RIP (Read Invalidate Pending) or OIP (Oread Invalidate Pending) is set in the fill CAM entry having the matching address, so that execution of the cache coherency transaction is deferred until the entry is removed from the fill CAM, as described above for handling a cache coherency transaction upon a cache block having an outstanding fill. If a fill CAM entry is for a READ LOCK, then the entry is not removed from the fill CAM until the corresponding WRITE UNLOCK is executed. Therefore, a cache coherency transaction deferred by the READ LOCK is not executed until the corresponding WRITE UNLOCK is executed.

Detailed Description Text (120):
A description of the fields in the CEFSTS register 308 is given in TABLE I. Each field is either a type WC (write-to-clear) or type RO (read-only). When a problem related to an outstanding fill occurs, the CEFSTS register holds information related to the problem, and the CEFADR register 307 holds the cache block address of the outstanding fill. If an outstanding fill times out or is terminated with RDE, the CEFADR register 307 and the CEFSTS register 308 are loaded and locked. The CEFADR register 307 is a read-only register.

Detailed Description Text (126):
RIP (Read Invalidate Pending) is set when a cache coherency transaction due to a read on the CPU bus is requested for a block which has Oread fills outstanding at the time. This triggers a writeback of the block when the fill data arrives; a valid

copy of the data is kept in the back-up cache 15.

Detailed Description Text (127):
OIP (Oread Invalidate Pending) is set when a cache coherency transaction due to an OREAD or a WRITE on the CPU bus is requested for a block which has OREAD fills outstanding at the time. This triggers a writeback and invalidate of the block when the fill data arrives.

Detailed Description Text (128):
DNF (Do Not Fill) is set when data for a read is not to be written into the backup cache 15. This is the case when the cache is off, in ETM, or when the read is to I/O space. The assertion of this bit prevents the block from being validated in the back-up cache.

Detailed Description Text (129):
RDLK.sub.-- FL.sub.-- DONE is set in the fill CAM when a READ LOCK hits in the back-up cache 15 or the last fill arrives from the BIU for a READ.sub.-- LOCK. Once this is set, the corresponding WRITE.sub.-- UNLOCK is allowed to proceed. This overrides the fill CAM block conflict on the WRITE UNLOCK which is inevitable since the READ.sub.-- LOCK is held in the fill CAM until the WRITE UNLOCK is executed.

Detailed Description Text (145):
Normally, the C-box controller (306 in FIG. 4) processes the read and write requests in the following order: first, any request in the D-read latch 299; second, any request in the I-read latch 300; and third, any request at the head of the write queue 60. Data reads are given priority over instruction reads in order to minimize the stalling of instruction execution for need of data by an already-decoded instruction. Reads should be given priority over writes because a read is needed for the current processing of an instruction, whereas a write is the result of already executed instruction. However, a read which follows a write to the same hexaword (i.e., the same cache block) should not be executed before the write, or else the read might return "stale" data.

Detailed Description Text (172):
To detect when there is a write-read conflict with a data stream read and the entries of the write queue, the write queue includes an address comparator 455 for each entry. AND gates 456, 457 set the DWR conflict bit for the entry when there is a hexaword address match and the entry is valid and the data stream read just occurred during the current clock cycle (when a signal NEW D-READ is asserted), so long as the entry is not also removed at the end of the current clock cycle.

Detailed Description Text (173):
The presence of a READ LOCK, and IPR READ, and a D-stream I/O command is detected by decoding logic 458. AND gates 459, 460 set the DWR conflict bit when the entry is valid and such a command just occurs during the current clock cycle (when the signal NEW D-READ is asserted), so long as the entry is not also removed at the end of the current clock cycle.

Detailed Description Text (174):
The presence of an I/O space write, an IPR WRITE, or a WRITE UNLOCK in the entry is detected by decoding logic 461. AND gates set the DWR conflict bit when the entry is valid and a D-read command just occurs during the current clock cycle (when the signal NEW D-READ is asserted), so long as the entry is not also removed at the end of the current cycle. To eliminate the decoding logic 461, however, the command codes for the write commands could be selected so that the presence of such a command is indicated by the state of a particular one of the five command bits CMD.

Detailed Description Text (183):
Preferably, the tag store control 473 is the state machine which executes any of the following tasks, upon instruction from the arbiter 471: TAG.sub.-- DREAD (performs a look-up for a data-stream read; and hits if the tag matches and is valid); TAG.sub.-- IREAD (performs a look-up for an instruction-stream read, hits if the tag matches and is valid, and may be cancelled midstream if the IREAD is aborted by the memory management unit); TAG.sub.-- OREAD (performs a look-up which requires ownership, and hits if the tag matches and the block is valid and owned); TAG.sub.--

R.sub.-- INVAL (performs a cache coherency look-up in response to a DREAD or IREAD from the CPU bus, and clears OWNED, if necessary); TAG.sub.--O.sub.-- INVAL (performs a cache coherency look-up in response to an OREAD or WRITE from the CPU bus, and clears VALID and/or OWNED, if necessary); TAG.sub.-- FILL (sets the VALID and/or OWNED bit for a fill which has been completed); IPR.sub.-- DEALLOC.sub.-- WRITE (performs a look-up for a de-allocate; clears VALID and OWNED bits if the block was owned); IPR.sub. -- TAG.sub.-- WRITE (writes the tag store with given data); and IPR.sub.-- TAG.sub.-- READ (reads the tag store from the location requested). When the tag store control 473 has finished executing a task, the tag store control notifies the arbiter 471.

Detailed Description Text (186):
In a first step 481, the arbiter gives highest priority to performing a de-allocate caused by a previous task. When a transaction such as a read miss causes a cache block to be de-allocated, this de-allocate always takes place in step 482 as the next data RAM task. In step 483, transactions in the in queue 61 are given the next-highest priority. Fills and cache coherency requests both arrive in the in queue 61, and then in step 484, the fill or cache coherency transaction at the head of the in queue is performed.

Detailed Description Text (191):
Turning now to FIG. 19, there is shown a flow chart of the steps followed by the arbiter 471 in servicing the D-read latch 299, the I-read latch 300, and the write queue 60. In steps 501, 502, 503, the source given priority asserts the address of its memory command upon the internal address bus 288 of the cache controller (see FIG. 4). If the memory command accesses an internal processor register (IPR) or I/O or a write unlock, as tested in step 504, then the command is completed in step 505. (To simplify implementation, the test in step 504 can be done concurrently with step 506 so that the fill CAM is always addressed and a hit always causes execution of a command other than a WRITE UNLOCK to stall.) If, however, the memory command accesses memory space, then in step 506, processing of the task is halted if there is a hit in the fill CAM. In this case, the memory space access conflicts with an outstanding fill or an outstanding READ LOCK. If, however, there is not a conflict with an outstanding fill or READ LOCK, then in step 507, execution branches depending on whether the cache is in the above-described error transition mode or whether the memory access is requested by an ownership command. If so, then in step 508, the tag RAMs are accessed to determine whether there is a cache hit in an owned block. If not, then if the cache is in the error transition mode, as tested in step 509, the back-up cache is bypassed and the read or write is sent directly to system memory (12 in FIG. 1) in step 510. If, however, in step 509, the cache was not operating in the error transition mode, then in step 511, an ownership read is sent to memory, and in step 512, the fill CAM is set to record that the refill is in progress. Moreover, if the addressed block in the cache is owned, as tested in step 513, then in step 514, the cache block is de-allocated and written back to memory in the next task. In other words, in step 514, a flag is set which is inspected by the arbitrator in step 481 of FIG. 18 to determine whether a need to de-allocate was caused by the previous task.

Detailed Description Text (192):
If in step 508 there was a cache hit in an owned block of the back-up cache, then in step 515, execution branches depending on whether the cache is in the error transition mode. If so, then in step 516, an ownership transaction is sent to memory, and the memory block is de-allocated and written back to memory in the next task. From step 515 or 516, execution continues in step 517 to complete the command.

Detailed Description Text (193):
If in step 507 it was found that the cache was neither in the error transition mode nor the command was an ownership command, then in step 518, execution branches depending on whether there was a cache hit. If so, then the command is completed in step 517. If not, then execution branches to step 519, where a refill of the cache block is begun by sending a data read or instruction read to memory. The fact that the refill is in progress is recorded in the fill CAM in step 512, and if the address block in the cache is owned, as tested in step 513, then in step 514, the address block is de-allocated and written back to memory in the next task.

Detailed Description Text (194):
Turning now to FIGS. 20A and 20B, there is shown a flow chart of the basic procedure followed by the arbiter 471 when servicing the in queue 61 of FIG. 4. In the first step 531 of FIG. 20A, the address of the in queue is asserted on the internal address bus (288 of FIG. 4) of the back-up cache controller. In step 532, execution branches depending on whether there is a fill CAM hit. If not, then execution branches to step 533 to determine whether the address of the transaction hits in the cache. If not, then the transaction is not pertinent to the back-up cache, and the end of the task is reached. Otherwise, then in step 534, an invalidate or write-back operation is performed upon the addressed cache block in accordance with Table B at the end of the specification, with any data RAM access for the deallocate and writeback being performed in the next task selected by the arbiter (471 in FIG. 17).

Detailed Description Text (195):
If in step 532 a fill CAM hit was found, then in step 535, execution branches depending upon whether the transaction is the return of read data. If not, then the transaction is an invalidate for the hit entry in the fill CAM. If the invalidate is an ownership read transaction, as tested in step 536, then in step 537, the OIP bit is set in the hit fill CAM entry, and execution of the transaction is finished for the current cycle, but will be completed later when the conflicting read lock or outstanding fill is completed. If in step 536 the invalidate was not an ownership read transaction, then it is a simple read invalidate. In step 538, execution of the transaction is finished if the addressed cache block is not owned, as indicated by the OREAD bit in the hit fill CAM entry. If the OREAD bit is set for the hit fill CAM entry, then in step 539, the RIP bit is set in the hit fill CAM entry so that the read invalidate transaction will be completed after the conflicting read lock or refill is finished.

Detailed Description Text (196):
If step 535 determines that the transaction is a return of read data, then in step 540, execution branches if the "do not fill" (DNF) bit is set in the hit fill CAM entry. If so, then in step 541 of FIG. 20B, the hit fill CAM entry is updated, and, in particular, if the transaction returns the last quadword of a fill, the fill CAM is cleared in step 541. If in step 540 the DNF bit is not set in the hit fill CAM entry, then in step 542 of FIG. 20B, the fill is written (or merged for a write operation) into the cache, and if the fill is for a read operation, then a specified portion of the fill may be transferred to the memory management unit. In step 543, execution branches depending on whether the fill is complete. If not, then in step 541, the hit fill CAM entry is updated and the task is completed. If so, then execution branches in step 555 depending on whether the OIP bit is set in the hit fill CAM entry. If so, then in step 556, execution branches depending on whether the OREAD bit is set in the hit fill CAM entry. If so, then in step 557, the cache block addressed by the address in the fill CAM entry is de-allocated and written back in the next task. Execution continues from step 556 or 557 in step 558, where the cache block addressed by the fill CAM entry is invalidated. Finally, in step 559, the hit fill CAM entry is cleared, and the task is finished.

Detailed Description Text (197):
If in step 555 it was found that the OIP bit was not set in the hit fill CAM entry, then execution branches to step 560 to test the RIP bit. If the RIP bit in the hit fill CAM entry is set, then in step 561, the cache block addressed by the fill CAM address is de-allocated and written back in the next task. Execution continues from step 560 or 561 in step 559, where the hit fill CAM entry is cleared, and the task is finished.

Detailed Description Text (198):
It should be appreciated that the control sequences in FIGS. 18-20B assume that various resources are available in the back-up cache controller for performing a selected task. If the required resources are not available, then a next lowest priority task may be performed if resources are available for performing that next-lowest priority task. In particular, the necessary conditions before servicing a fill from the in queue 61 are: (1) the data RAMs and the tag store must be free, and (2) if RIP or OIP is set in either fill CAM entry, the write-back queue 63 must

not be full, because a write-back may be necessary at the completion of the fill. Necessary conditions before servicing a cache coherency request from the in-queue 61 are that the tag store must be free. If the cache coherency request hits owned and requires a write-back, and the write-back queue 63 is full, then the cache coherency request is stalled until the write-back queue is no longer full. Necessary conditions before servicing a command from the D-read latch 299 or the I-read latch 300 are: (1) the data RAMs and the tag store must be free; (2) a fill CAM entry must be available, in case the read misses; (3) there must be an available entry in the non-write-back queue 62, in case the read misses; (4) there must be no valid entry in the fill CAM for the same cache block as that of the new request; (5) there must be no RDLK bits set in the fill CAM, indicating that a READ LOCK is in progress; and (6) there must be no block conflict with any write queue entry. If a read misses owned and requires a de-allocate, and the write-back queue 63 is full, then the read is stalled until the write-back queue is no longer full. Necessary conditions before servicing a full quadword write from the write queue 60 are: (1) the tag store must be free; (2) a fill CAM entry must be available, in case the write misses and requires an OREAD; (3) there must be an available entry in the non-write-back queue 62, in case the write misses; (4) there must be no valid entry in the fill CAM for the same cache block as that of the new request; and (5) if there is a READ LOCK in the fill CAM, the fills for the READ LOCK must have completed. If the full quadword write misses owned and requires a deallocate, and the write-back queue 63 is full, the quadword write is stalled until the WRITE BACK queue is no longer full.

Detailed Description Text (199):
Preferably, the tag store look-up for a full quadword write may be done while the data RAMs are busy with another transaction. When the data RAMs free up, the full quadword write is done. If the full quadword writes are streaming through the write queue 60, this effectively pipelines the tag store access and the data RAMs accesses so that the write takes place at the maximum write repetition rate of the data RAMs. This would not be the case if the arbiter required both the data RAMs and the tag store to be free before starting the full quadword write.

Detailed Description Text (200):
Necessary conditions before servicing any write queue entry other than a full quadword write are as follows: (1) the tag store and the data RAMs must be free; (2) a fill CAM entry must be available, in case the write misses and requires an OREAD; (3) there must be an available entry in the non-write-back queue 62, in case the write misses; (4) there must be no valid entry in the fill CAM for the same cache block as that of the new request; (5) if there is a READ LOCK in the fill CAM, the fills for the READ LOCK must have completed; and (6) if the write queue entry is a write unlock or an IPR write, there must be an available entry in the write-back queue. If a write misses owned and requires a de-allocate, and the write-back queue is full, the write is stalled until the write-back queue is no longer full.

Detailed Description Paragraph Table (1):
TABLE A Backup Cache Block State VALID=0, OWNED=0 invalid block VALID=0, OWNED=1 invalid block (this combination of state bits should never happen) VALID=1, OWNED=0 valid block (also referred to as valid-unowned) VALID=1, OWNED=1 owned block (also referred to as valid-owned)

Detailed Description Paragraph Table (2):
TABLE B Normal Backup Cache Behavior cache state of the block in the cache coherency Invalid Valid, valid, command block unowned block owned block IREAD, no action no action writeback, set block DREAD state to valid-unowned OREAD, no action invalidate writeback, invalidate WRITE WDISOWN no action no action no action

Detailed Description Paragraph Table (3):
TABLE C CPU Bus Command Encodings and Definitions Command Bus Field Abbrev. Transaction Type Function 0000 NOP No Nop No Operation Operation 0010 WRITE Write Addr Write to memory with byte enable if quadword or octaword 0011 WDISOWN Write Addr Write memory; Disown cache disowns block and returns ownership to

memory 0100 IREAD Instruction Addr Instruction- Stream stream read Read 0101 DREAD Data Addr Data-stream Stream read (without Read ownership) 0110 OREAD D-Stream Addr Data-stream Read read claiming Ownership ownership for the cache 1001 RDE Read Data Data Used instead Error of Read Data Return in the case of an error. 1010 WDATA Write Data Data Write data Cycle is being transferred 1011 BADWDATA Bad Write Data Write data with Data error is being transferred 1100 RDRO Read Data0 Data Read data is Return(fill) returning corresponding to QW 0 of a hexaword. 1101 RDR1 Read Data1 Data Read data is Return(fill) returning corresponding to QW 1 of a hexword. 1110 RDR2 Read Data2 Data Read data is Return(fill) returning corresponding to QW 2 of a hexaword. 1111 RDR3 Read Data3 Data Read data Return(fill) is returning corresponding to QW 3 of a hexaword.

Detailed Description Paragraph Table (4):
TABLE D Backup Cache Behavior During Error Transition Mode (ETM) Cache Cache Response Transaction Miss Valid hit Owned hit CPU IREAD, Read from Read from Read from DREAD memory memory cache Read Modify CPU Read from Read from Force block READ.sub.-- LOCK memory memory writeback, read from memory CPU WRITE Write to Write to Force block memory memory write back, write to memory CPU Write to Write to Write to WRITE.sub.-- UNLOCK memory memory cache Fill (from Normal cache behavior read started before ETM) Fill (from Do not update backup cache; read started return data to Mbox during ETM) NDAL cache Normal cache behavior* coherency request *Except that cache coherency transaction due to ORead or Write always results in an invalidate to PCache, to maintain PCache coherency whether or not BCache hit, because PCache is no longer a subset

Detailed Description Paragraph Table (5):
TABLE E Backup Cache State Changes During ETM Cache Cache State Modified Transaction Miss Valid hit Owned hit CPU IREAD,DREAD None None None Read Modify CPU READ.sub.-- LOCK None None Clear VALID & OWNED; change TS.sub.-- ECC accordingly. CPU Write None None Clear VALID & OWNED; change TS.sub.-- ECC accordingly. CPU None None Write new data, WRITE-UNLOCK change DR.sub.-- ECC accordingly. Fill (from read .sub.-- Write new TS.sub.-- TAG, TS.sub.-- VALID, started TS.sub.-- OWNED, TS.sub.-- ECC, DR.sub.-- DATA, before ETM) DR.sub.-- ECC.sub.-- Fill (from read started None during ETM) NDAL cache coherency Clear VALID & OWNED; change request TS.sub.-- ECC accordingly

Detailed Description Paragraph Table (6):
TABLE F Contents of a Fill CAM Entry ADDRESS<31:3> Quadword-aligned address of the read request. RDLK Indicates that a READ LOCK is in progress. IREAD This is an Istream read from the Mbox which may be aborted. OREAD This is an outstanding OREAD; block ownership bit should be set when the fill returns. WRITE This read was done for a write; write is waiting to be merged with the fill. TO.sub.-- MBOX Data is to be returned to the Mbox. RIP READ invalidate pending. OIP OREAD invalidate pending. DNF Do not fill - data is not to be written into the cache or validated when the fill returns. RDLK.sub.-- FL.sub.-- DONE Indicates that the last fill for a READ.sub.-- LOCK arrived. REQ.sub.-- FILL.sub.-- DONE Indicates that the requested quadword of data was received from the CPU bus. COUNT<1:0> Counts the number of fill quadwords that have been successfully returned. VALID Indicates that the entry contains valid information.

Detailed Description Paragraph Table (8):
TABLE H Control by RIP/OIP After a Fill Action taken immediately State of RIP/OIP after the fill is complete RIP=0, OIP=0 no cache coherency action taken RIP=0, OIP=1 Oinval initiated to the cache block which was just filled; the block is written back to memory if the access resulted in hit-owned; both VALID and OWNED are cleared. RIP=1, OIP=0 Rinval initiated to the cache block which was just filled; the block is written back to memory; OWNED is cleared. The block remains valid in the cache (i.e., that CPU has read privileges to that block, but does not have write privileges). RIP=1, OIP=1 if this state occurs, it is treated the same as for RIP=0, OIP=1

Detailed Description Paragraph Table (9):
TABLE I _____ CEFSTS Field Descriptions Name Extent Type Description _____ RDLK 0 WC Indicates that a READ.sub.-- LOCK was in progress. LOCK 1 WC Indicates that an error occurred and the register is locked. TIMEOUT 2 WC FILL failed due to transaction timeout. RDE 3 WC FILL failed due to Read Data Error. LOST.sub.-- ERR 4 WC Indicates that more than one error related to fills occurred. IDO 5 RO CPU bus identification bit for the read request. IREAD 6 RO This is an Istream read from the Mbox which may be aborted. OREAD 7 RO This is an outstanding OREAD. WRITE 8 RO This read was done for write. TO.sub.-- MBOX 9 RO Data is to be returned to the Mbox. RIP 10 RO READ invalidate pending. OIP 11 RO OREAD invalidate pending. DNF 12 RO Do not fill - data not to be written into the cache or validated when the fill returns. RDLK.sub.-- FL.sub.-- DONE 13 RO Indicates that the last fill for a READ-LOCK arrived. REQ.sub.-- FILL.sub.-- DONE 14 RO Indicates that the requested quadword was successfully returned from the CPU bus. COUNT 16:15 RO For a memory space transaction, indicates how many of the fill quadwords have been successfully returned. For I/O space, is set to 11(BIN) when the transaction starts as only one quadword will be returned. UNEXPECTED.sub.-- FILL 21 WC Set to indicate that an unexpected fill was received from the CPU bus. _____

CLAIMS:

1. A method of operating a first processor in a multi-processor digital computer system having said first processor, a second processor, and a system memory accessed by both of said first and second processors over a system bus operating in accordance with a block ownership cache coherency protocol, said first processor having a cache memory for storing blocks of data in association with memory addresses, said method comprising the steps of:

a) fetching data having a specified memory address for a data processing operation by searching said cache memory for said specified memory address, and when said specified memory address is not found in said cache memory, storing the specified memory address in a content addressable memory, and sending a fill data request including said specified memory address to the system memory;

b) before receipt of fill data from the system memory,

i) receiving a cache coherency request from said second processor in accordance with said block ownership cache coherency protocol, said cache coherency request including said specified memory address and requesting invalidation of a block of data having the specified memory address, and

ii) checking whether said specified memory address is stored in said content addressable memory, delaying execution of said cache coherency request until said fill data is received from said system memory; and

c) receiving said fill data from said system memory, and using said fill data for said data processing without retaining a validated block of said fill data in said cache memory.

3. The method as claimed in claim 1, wherein said cache coherency request is a read invalidate request requesting said first processor to relinquish any exclusive privilege to write to said block of data having said specified memory address, and wherein said method includes clearing an indication of ownership associated with said block of said fill data in said cache memory so that a block of said fill data validated for writing is not retained in said cache memory, and writing said block of said fill data in said cache memory back to said system memory.

4. The method as claimed in claim 3, wherein said data processing operation includes the writing of data to said specified memory address, and wherein said writing of said block of said fill data in said cache memory back to said system memory writes the data written in accordance with said data processing operation back to said system memory.

5. The method as claimed in claim 1, wherein said step of delaying execution of said cache coherency request includes setting an indication associated with said specified memory address in said content addressable memory so as to indicate a delayed invalidate operation, and upon receiving said fill data for said specified memory address from said system memory, checking whether a delayed invalidate operation is indicated in association with said specified memory address in said content addressable memory, and upon finding that a delayed invalidate operation is indicated, not retaining a validated block of said fill data in said cache memory.

6. The method as claimed in claim 5, further comprising the step of loading said fill data into a cache block in said cache memory, and invalidating the cache block into which said fill data was loaded so as not to retain a validated block of said fill data in said cache memory.

7. The method as claimed in claim 1, wherein said cache coherency request is an ownership invalidate request requesting said first processor to refrain from writing to or reading from said block of data having the specified memory address, and wherein said method includes the step of clearing an indication of validity associated with said block of said fill data in said cache memory.

8. The method as claimed in claim 7, wherein said data processing operation includes the writing of data to said specified memory address, said fill data request includes a request for an exclusive privilege to write to said block of data having said specified memory address, and wherein said method further includes writing said block of said fill data in said cache memory back to said system memory.

9. The method as claimed in claim 8, wherein said data processing operation includes the writing of data to said specified memory address, and wherein said writing of said block of said fill data in said cache memory back to said system memory writes the data written in accordance with said data processing operation back to said system memory.

10. A method of operating a first processor in a multi-processor digital computer system having said first processor, a second processor, and a system memory accessed by both of said first and second processors over a system bus operating in accordance with a block ownership cache coherency protocol, said first processor having a cache memory for storing blocks of data and a memory address associated with each of said blocks of data, said method comprising the steps of:

a) fetching data having a specified memory address for a data processing operation by said first processor by searching said cache memory for said specified memory address, and when said specified memory address is not found in said cache memory, storing the specified memory address in an entry of a content addressable memory having a plurality of entries, and sending a fill data request including said specified memory address to said system memory;

b) before receipt of fill data from the system memory,

i) receiving a cache coherency request from said second processor in accordance with said block ownership cache coherency protocol, said cache coherency request including said specified memory address, and

ii) addressing said content addressable memory with the specified memory address of said cache coherency request, and upon finding that the specified memory address of said cache coherency request is in said entry of said content addressable memory, setting in said content addressable memory an indication that said cache coherency request is pending for said specified memory address; and

c) receiving said fill data from said system memory, using said fill data for said data processing operation by said first processor, checking said entry of said content addressable memory for said indication that said cache coherency request is pending for said specified memory address, executing said cache coherency request.

11. The method as claimed in claim 10, wherein said cache coherency request is an ownership-read invalidate request requesting said first processor to refrain from

reading or writing to a block of data having said specified memory address, said method includes storing said fill data in a cache block in said cache memory, and wherein the execution of said cache coherency request includes clearing an indication of validity associated with said cache block in said cache memory.

12. The method as claimed in claim 10, wherein said data processing operation includes the writing of data to said specified memory address, said fill data request includes a request for an exclusive privilege to write to said specified memory address, said method includes setting in said entry of said content addressable memory an indication of said request for an exclusive privilege to write to said specified memory address, said cache coherency request is a read invalidate request requesting said first processor to relinquish any exclusive privilege to write to a block of data having said specified memory address, said step (b) (ii) further includes checking whether said entry indicates said request for an exclusive privilege to write to said specified memory address, and wherein the setting in said content addressable memory of an indication that said cache coherency request is pending for said specified memory address is performed upon finding that said entry indicates said request for an exclusive privilege to write to said specified memory address.

13. The method as claimed in claim 10, wherein said data processing operation includes the writing of data to said specified memory address, said fill data request includes a request for an exclusive privilege to write to said specified memory address, said method includes setting in said entry of said content addressable memory an indication of said request for an exclusive privilege to write to said specified memory address, said cache coherency request is an ownership-read invalidate request requesting said first processor to refrain from reading or writing to a block of data having said specified memory address, and wherein step (c) further includes checking whether said entry indicates said request for an exclusive privilege to write to said specified memory address, and upon finding that said entry indicates said request for an exclusive privilege to write to said specified memory address, writing a block of fill data including data written in accordance with said data processing operation back to said system memory.

14. The method as claimed in claim 10, wherein said cache coherency request is a read invalidate request requesting said first processor to relinquish any exclusive privilege to write to a block of data having said specified memory address, said method includes storing said fill data in a cache block in said cache memory, and wherein the execution of said cache coherency request includes clearing an indication of ownership associated with said cache block in said cache memory storing said fill data, and writing said cache block of fill data back to said system memory.

15. The method as claimed in claim 14, wherein said data processing operation includes the writing of data to said specified memory address, and wherein said writing of the cache block back to said system memory writes the data written in accordance with said data processing operation back to said system memory.

16. A processor for a multi-processor computer system, said multi-processor computer system having a system bus for coupling processors to a system memory, said system bus operating in accordance with a block ownership cache coherency protocol, said processor comprising, in combination:

instruction decoding means for decoding computer program instructions to generate requests for reading data at specified read addresses;

instruction execution means connected to said instruction means for executing the computer program instructions decoded by said instruction decoding means to generate requests for writing data at specified write addresses;

a cache memory for storing blocks of data, and in association with each block of data, a memory address, an indication of whether each block is valid for providing data from said memory address in response to said requests for reading data, and an indication of whether each block is valid for receiving data from said requests for writing data to said memory address;

a content addressable memory including a plurality of entries and means for storing in each entry a fill address of a fill request to a system memory in said multi-processor system requesting fill data from said fill address in said shared memory, an indication of whether the fill address is associated with a request for validation for writing data to said fill address, an indication of whether a read invalidate request was received, before return of said fill data, from another processor in said multi-processor system requesting invalidation of any indication that a cache block having said fill address in said cache memory is valid for receiving write data of whether an ownership-read invalidate request was received, before return of said fill data, from another processor in said multi-processor system requesting invalidation of any indication that a cache block having said fill address is valid for providing read data;

means, responsive to a request for reading data from a specified read address, for addressing said cache memory with said read address, for reading data from said cache memory when said cache memory contains a cache block having said read address and indicated as valid for providing read data, and when said cache memory does not contain a cache block having said read address and indicated as valid for providing read data, for sending a fill request to said main memory including said read address and for storing said read address in said content addressable memory;

means, responsive to a request for writing data to a specified write address, for writing data to said cache memory when said cache memory contains a cache block having said write address and indicated as valid for receiving write data, and when said cache memory does not contain a cache block having said write address and indicated as valid for receiving write data, for sending a fill request to said system memory including said write address and a request for validation for a write operation, and for storing in said content addressable memory said write address together with an indication that the fill address is associated with a request for validation for a write operation;

means, responsive to receiving from another processor in said multi-processor system a read invalidate request having a specified read invalidate address, for addressing said content addressable memory with said specified read invalidate address, and when a fill address matching said specified read invalidate address is found in said content addressable memory, for setting the indication of whether a read invalidate request was received from another processor in said multi-processor system before return of said fill data;

means, responsive to receiving from another processor in said multi-processor system an ownership-read invalidate request having a specified ownership-read invalidate address, for addressing said content addressable memory with said specified ownership-read invalidate address, and when a fill address matching said specified ownership-read invalidating address is found in said content addressable memory, for setting the indication of whether an ownership-read invalidate request was received from another processor in said multi-processor system before return of said fill data;

first means, responsive to return of said fill data for checking said indication in said content addressable memory of whether an ownership-read invalidate request was received before return of said fill data, and when an ownership-read invalidate request was received before return of said fill data, for invalidating an indication that a cache block having the fill address in said cache memory is valid for providing read data; and second means, responsive to return of said fill data, for checking the indication in said content addressable memory of whether a read invalidate request was received before return of said fill data, and when a read invalidate request was received before return of said fill data, for invalidating an indication that a cache block having the fill address in said cache memory is valid for receiving write data.

18. The processor as claimed in claim 16, wherein said first means responsive to return of said fill data includes means for checking whether the fill address of said fill data is associated with a request for validation for a write operation, and when the fill address of said fill data is associated with a request for

validation of a write operation and an ownership-read invalidate request was received before return of said fill data, filling a cache block with said fill data and writing back to said system memory data in the cache block having been filled with said fill data.

19. The processor as claimed in claim 18, further including writing write data of said write operation to said cache block having been filled with said fill data before writing back to the system memory said data in the cache block having been filled with said fill data.

WEST

Generate Collection Print

L30: Entry 4 of 8 File: USPT Jun 17, 1997

DOCUMENT-IDENTIFIER: US 5640563 A
TITLE: Multi-media computer operating system and method

Drawing Description Text (8):
FIG. 4B illustrates the portion of the task execution queue management process in which a given task or group of tasks having equal completion deadlines may be initiated for execution.

Drawing Description Text (9):
FIG. 5 illustrates the interrupt handling process in its effect on management of the idle task list and interaction with the active task execution queue.

Detailed Description Text (9):
Frame managers are ordered in the queue or schedule by the end times, i.e. the task completion deadlines, at the iteration interval at which the frame manager is invoked. The earliest task completion deadline or end time is the first priority entry and other frame managers are linked in a prioritized order based on their end times with the second earliest end time being the second entry, etc. Whenever two or more frame managers have equal ending times, as can result where the recurrent periods form multiples of one another, etc., the frame managers having equal task deadline completion times may be further ordered in the doubly linked list or prioritized therein by their starting times, with the earliest starting time task frame manager being first. In the doubly linked list, the first frame manager entry in the schedule is chosen to be a dummy frame manager called the "top of schedule" which is a place holder for containing the pointer to the first address in memory where the present highest priority frame manager resides. The last frame manager entry in the doubly linked list is always assigned to a non-real-time frame manager that is utilized for calling any non-real-time task, such as housekeeping tasks, that may be resident in the system and which are executed only after all of the hard, real-time tasks have been completed.

Detailed Description Text (10):
In the present embodiment, there is also an idle frame manager list or idle task queue. It contains the ordered list of all idle frame managers, i.e. those frame managers that have completed their execution for the current time and are awaiting their next recurring start times. The idle list entries are further arranged corresponding to various levels of interrupt sources, there being a separate list for each level of interrupt source. The frame manager idle lists are also doubly linked lists containing frame manager indicators awaiting the occurrence of their next starting or invocation time for execution. The interrupt handlers in the present invention are separate and keep track of the frame managers in the idle list corresponding to the level of interrupt on which a given task or frame manager resides. Frame managers in the idle list are ordered or prioritized by their start times rather than the task completion or end times, with the earliest start time being first. When two or more frame managers happen to have equal starting times, only the frame manager with the earliest end time is kept in the idle frame manager list.

Detailed Description Text (11):
A further, much more detailed description of the task execution queue of frame managers, of the queue or list of idle frame managers and of the various portions of control code and control data utilized will be described with reference to FIG. 1. However, in order to understand the advantage achieved by the present operating

system invention in prioritizing the queue of tasks for execution in accordance with the task completion deadlines, a further explanation follows.

Detailed Description Text (26):
Returning now to a more detailed description of the preferred embodiment, it may be noted that the tasks themselves are not directly scheduled and rescheduled. Instead, place holders which are called "frame managers" and which correspond to identifiers for invoking a common task handling code routine are inserted in the queue. The frame manager is, in fact, a special task itself and executes a group of common, periodically recurring tasks at a frequency of recurrence that matches that of any tasks associated with the frame manager. A frame manager executes all tasks having the same task completion deadline which may be assigned to it and returns to the operating system for rescheduling in the queue when it has completed all of its assigned tasks.

Detailed Description Text (28):
The frame control blocks are constructed by the host system and passed to the digital signal processor for storage as control data. The frame control block is a data structure that contains all of the information that the operating system of the present invention needs in order for it to schedule the frame manager into the execution queue or list. It also contains designation of the area or address in memory where the processor registers in the digital processor may be saved when task switching is executed. The FCB also contains the address designations for digital signal processor storage area or memory for any processing tasks to be associated with the frame manager and the identifying starting address for the beginning of the frame manager's linked list of subtasks having equal task completion deadline periods or times. A typical frame control block must be initialized as shown in Table 1, below, before the frame manager is activated.

Detailed Description Text (29):
The construction of an FCB is carried on by the host processor in conjunction with the DSP and is not specifically a part of this invention. Therefore only this resulting structure and content of the frame control block is shown in the table. In Table 1, the first two entries are the pointers to the beginning and ending of the idle frame manager list which is held in association with each interrupt level as control data. A third entry is an alarm or start time indicator and the fourth entry is the interrupt linking address pointer to the interrupt handler data structure associated with the given FCB. The fifth and sixth entries are the pointers to the beginning and end addresses of the schedule or task execution priority queue of frame managers. The 15th through 38th entries are the addresses in memory where context of the digital signal processor's current operating conditions may be saved whenever preemption of the frame manager occurs.

Detailed Description Text (37):
FIRSTTCB--This entry is an address of the beginning of the frame manager's linked list of tasks to be executed by that frame manager on or before the occurrence of its specific deadline at its recurrent frequency. This location is normally used to indicate a task control block (TCB) of the first task assigned to the frame manager. However, in order to allow a frame manager to be activated without any tasks being linked to it, this entry may be initialized to 0. Whenever this entry holds a 0, the frame manager will simply return to be rescheduled. Other data locations in the FCB are used to accommodate scheduling and task switching and are briefly described as follows.

Detailed Description Text (63):
Referring to FIG. 1, the operating system first accesses the frame manager scheduling code contained in the control code block 1 in memory. The processes are described in detail with relation to FIGS. 2 and 3. The operating system searches the idle list 2 associated with the interrupt level of a given piece of hardware or task being invoked. It then inserts an identifier for the specific frame manager being invoked into the idle list as is also described in detail in FIGS. 2 and 3. The operating system then searches the frame manager prioritized execution schedule 3, which resides as control data in memory, to find the proper starting point based on the task or frame manager end times. The operating system searches the frame manager priority execution schedule in control data memory to find a proper starting

point for insertion of a given FCB based on the end times of the FCB to be invoked relative to the end times of any other FCBs currently in the schedule. The operating system inserts an address pointer to the frame manager FCB in the control code in memory.

Detailed Description Text (64):
Next the operating system schedule scanner 4 in FIG. 1 searches the frame manager prioritized execution schedule 3 for the next frame manager to be executed. It then accesses the frame control block for the frame manager and finds the next task in that frame manager which is ready to run. The schedule scanner 4 passes the FCB of the frame manager to the frame manager initiation control 5 which checks whether the given frame manager is resuming after interruption or is beginning a new iteration. The frame manager 6 receives initiation from the frame manager initiation block 5 and calls the next task to be executed from among any task or tasks associated with the given frame control block. The initiation control 5 gets information of where to begin or to resume an interrupted task from the fields in the FCB.

Detailed Description Text (104):
It may be seen from the foregoing that FIG. 3 illustrates a process in which the prioritized queue of frame managers corresponding to user tasks to be executed is established and managed in accordance with the earliest frame manager deadline or end time. Any user tasks having the same frequency of recurrence and same required result available deadline must be handled by this frame manager during its execution period. The list or queue of frame managers in their prioritized order is sorted and rescheduled as shown whenever a frame manager end time is reached or a new frame manager is inserted by execution of the non-real-time frame manager code or deleted by such code.

Detailed Description Text (114):
Frame managers 11, 5 and 3 all have starting times beginning at interrupt 0. The order of the execution schedule, based on ending times will thus be frame manager 3, followed by 5 followed by 11. Frame manager 3, being the first frame manager in the execution schedule, will begin execution at interrupt 0. Execution of frame manager 3's task is shown to complete sometime after interrupt 1 and will thus be rescheduled to begin again at interrupt 3 with a new ending time at interrupt 6. The new order of execution of the schedule is now frame manager 5, followed by 3 followed by 11. Frame manager 5 will begin execution next because it has already reached a start time and it is now first in the execution schedule, frame manager 3 having been executed. Frame manager 5 will complete execution sometime after interrupt 2 has been reached and will be rescheduled to a new start time beginning at interrupt 5 with a new ending time at interrupt 10. The new order of the execution schedule will now be frame manager 3 followed by 5 followed by 11. However, at this point in time, i.e. just after interrupt 2, frame managers 3 and 5 proceed frame manager 11 in the execution schedule, but their starting times have not yet occurred. Frame managers 3 and 5 are thus bypassed and frame manager 11 actually begins execution.

Detailed Description Text (122):
The foregoing process will continue throughout many iterations until interrupt eighteen, when frame manager eleven will complete execution again. At this time frame managers three, five, and eleven have all completed execution for their current frame and their start times have not re-occurred; and when this condition prevails, the non-real time frame manager task may be permitted to execute. Eventually an interrupt twenty occurs and the start time of frame manager five will be reached and the non-real time task frame manager will be interrupted and execution of the frame manager five will resume as described above.

WEST

End of Result Set

Generate Collection Print

L31: Entry 1 of 1 File: USPT Nov 3, 1998

DOCUMENT-IDENTIFIER: US 5832484 A
TITLE: Database system with methods for parallel lock management

Abstract Text (1):
Database system and methods are described for improving scalability of multi-user database systems by improving management of locks used in the system. The system provides multiple server engines, with each engine having a Parallel Lock Manager. More particularly, the Lock Manager decomposes the single spin lock traditionally employed to protect shared, global Lock Manager structures into multiple spin locks, each protecting individual hash buckets or groups of hash buckets which index into particular members of those structures. In this manner, contention for shared, global Lock Manager data structures is reduced, thereby improving the system's scalability. Further, improved "deadlock" searching methodology is provided. Specifically, the system provides a "deferred" mode of deadlock detection. Here, a task simply goes to sleep on a lock; it does not initiate a deadlock search. At a later point in time, the task is awakened to carry out the deadlock search. Often, however, a task can be awakened with the requested lock being granted. In this manner, the "deferred" mode of deadlock detection allows the system to avoid deadlock detection for locks which are soon granted.

Brief Summary Text (9):
One area where system performance sometimes has been found to be lacking is the server functionality responsible for coordinating access of multiple clients to shared database objects, specifically the aspect of "locking" database objects. Locking mechanisms are employed on DBMSs to prevent multiple tasks from concurrently sharing a resource (e.g., reading or writing a database record) in a manner which leads to inconsistency of the resource. Common concurrency problems include lost updates, uncommitted dependencies, and inconsistent analyses. These problems and some locking mechanisms designed to address them are described in the general database literature; see, e.g., Nath, A., The Guide to SQL Server, Second Edition, Addison-Wesley Publishing Company, 1995, which is hereby incorporated by reference for all purposes.

Brief Summary Text (10):
Briefly, certain tasks require "exclusive" access, while other tasks need only "shared" access. When a client accesses a database record for the purpose of updating that record, the client must take out an exclusive "lock" on that record to ensure that while the task is using the record, another task cannot concurrently modify (and in some cases also not read) the record. For example, if a first user updates a data record to reflect a recent transaction, a lock on that record assures the user that no other users can concurrently modify the record while his or her transaction is being processed.

Brief Summary Text (11):
In one approach to lock management, such as found in Sybase SQL Server.TM. (System 10), different locks are provided in shared memory and are made available by a global resource manager commonly referred to as a lock manager. When a task requires access to a record, the lock manager determines whether an appropriate lock can be granted, and, if so, allocates such a lock to the task. If the lock manager finds an existing lock on the record, it either queues a request for the lock or grants a concurrent lock to the task, depending upon the nature of the existing lock and the

type of access required (i.e., whether the locks can co-exist).

Brief Summary Text (19):
The Lock Manager provides an internal application programming interface ("API") to its internal clients, such as the Access Methods. Because the Lock Manager is a global resource, identical copies are provided in each engine of a database network (as identical pieces of server code). In particular, an SMP (symmetric multi-processor) embodiment of the Database Server System supports multiple database engines (i.e., multiple instances of the engine), up to an arbitrary number of database engines. Each engine includes a Lock Manager instance. Even though there are multiple instances of the Lock Manager, the engines share a single, global set of data structures.

Brief Summary Text (20):
Conventionally, such structures were protected by a single spin lock. Such an approach robs a system's ability to scale in an SMP environment, due to contention among the multiple server engines for the single spin lock. To reduce this contention for these shared data structures, therefore, the present invention provides the Lock Manager (of each engine) with methods for parallel lock management--decomposing the (prior) single spin lock into multiple spin locks, each protecting individual hash buckets or groups of hash buckets.

Brief Summary Text (22):
Access to particular lock types is controlled via hash tables, one table for each lock type. Each hash table has one or more spin locks, each spin lock for controlling a subset or range of one or more hash buckets for the table. In this manner, contention is substantially reduced: locks only block when two tasks each concurrently require access to hash buckets covered by the same spin lock. In previous systems, in contrast, a single spin lock was always used to protect such structures and, thus, always presented a bottleneck.

Brief Summary Text (23):
By associating groups of hash buckets with a particular spin lock, spin locks in the system of the present invention are conserved. The number of spin locks employed for each hash table is user (administrator) configurable. The simplest use is to configure the system such that the minimum number of spin locks is employed: one spin lock for each hash table. Even at this minimum, contention is reduced: processing a lock request of one type of lock (e.g., page lock) will not block access to other types of locks (e.g., address locks). At the other extreme, the user can allocate one spin lock for each and every bucket (up to the number of spin locks available in the system). Here, increasing the number of spin locks increases parallelism available with a give lock type--contention for other hash buckets within a given table decreases as the number of spin locks assigned to that table increases. In typical use, the user will configure or tune use of available spin locks to optimize system performance. In this manner, contention for access to the Lock Manager can be reduced while preserving efficient use of system spin locks, thereby allowing the system to appropriately scale up as more engines are added to the system.

Drawing Description Text (2):
FIG. 1A is a block diagram illustrating a computer system in which the present invention may be embodied.

Drawing Description Text (3):
FIG. 1B is a block diagram illustrating a software subsystem for controlling the operation of the computer system of FIG. 1A.

Drawing Description Text (4):
FIG. 2A is a block diagram of a client/server system in which the present invention may be embodied.

Drawing Description Text (5):
FIG. 2B is a block diagram of an SMP client/server system in which the present invention may be preferably embodied.

Drawing Description Text (6):
FIG. 2C is a block diagram illustrating shared Lock Manager structures in the client/server system of FIG. 2B which are, before modification in accordance with the present invention, under control of a single spin lock.

Drawing Description Text (7):
FIG. 3A is a block diagram of a Lock Manager illustrating shared Lock Manager structures which have been modified in accordance with the present invention.

Drawing Description Text (8):
FIG. 3B is a block diagram illustrating layout and use of hash tables provided by the shared Lock Manager structures.

Drawing Description Text (9):
FIG. 4A is a diagram illustrating use of data structures "hanging" off a hash table entry ("hash bucket") for representing a request for an exclusive lock.

Drawing Description Text (10):
FIG. 4B is a diagram illustrating use of data structures for representing multiple requests for exclusive locks.

Drawing Description Text (11):
FIG. 4C is a diagram illustrating use of data structures for representing multiple requests for shared locks.

Drawing Description Text (12):
FIG. 4D is a diagram illustrating use of data structures for representing multiple requests for both shared and exclusive locks.

Drawing Description Text (13):
FIG. 4E is a diagram illustrating addition of a new request for an exclusive lock.

Drawing Description Text (14):
FIG. 4F is a diagram illustrating addition of a new request for a shared lock.

Drawing Description Text (21):
FIG. 9 is a block diagram illustrating a method of the present invention for improved management of global "free" locks.

Detailed Description Text (5):
A "database object" is an article stored in one or more databases. Examples of such database objects include tables, tuples, logs for databases, statistics pages, views, stored procedures, and the like. Such database objects may have attributes such as sizes, names, types, and so forth. In addition, a database object is a "resource" employed by the database during normal functioning. Examples of such resource-type database objects include buffers and semaphores. In the context of this invention, locks can be placed on some or all of these database object types.

Detailed Description Text (6):
A "semaphore" is an indicator, such as a flag, used to govern access to shared system resources, for maintaining order among processes that are competing for use of resources.

Detailed Description Text (11):
Transactional locks, or simply "locks," are employed by the system to coordinate OLTP (on-line transactional processing). Typical transactional locks include, for instance, a table lock and a page lock. These locks can be shared or exclusive. An "exclusive" lock, when held by a given process, completely excludes all other processes from accessing the locked object. A "non-exclusive" or "shared" lock, on the other hand, permits certain types of shared access while the lock is held by a given process. For example, if a first process takes out a non-exclusive lock on a data page, a second process may be permitted to read data records from that page even while the first process holds the lock. However, the non-exclusive lock might prevent the second process from writing to any data records contained in the locked data page.

Detailed Description Text (13):
A task or context switch itself is not cost-free: system resources are required for swapping out the context of one task for another. For instance, the then-current state of the task to be put to sleep must be stored, while the system switches to another task. Once the task is awakened, the system must restore the state information for the awaking task. For certain system critical resources, the overhead incurred with such a task switch is unacceptable. By spinning instead of sleeping, a task can avoid the overhead and cost of a context switch. To ensure that waiting tasks do not spin too long, spin locks typically contain only very few instructions, so that they are held for only a very short duration. Spin locks are employed within the system of the present invention to control access to low-level system resources, such as important lock management data structures which are shared.

Detailed Description Text (18):
Also shown, the software system 150 includes a Relational Database Management System (RDBMS) front-end or "client" 170. The RDBMS client 170 may be any one of a number of database front-ends, including PowerBuilder.TM., dBASE.RTM., Paradox.RTM., Microsoft.RTM. Access, or the like. In an exemplary embodiment, the front-end will include SQL access drivers (e.g., Borland SQL Links, Microsoft ODBC drivers, Intersolv ODBC drivers, and the like) for accessing SQL database server tables in a Client/Server environment.

Detailed Description Text (33):
Of particular interest herein is an embodiment of the present invention in the Client/Server System 200 operating on an symmetric multi-processor (SMP) computer system. Such an embodiment is shown in FIG. 2B, as Client/Server System 200a. The system includes an SMP server 230a providing an SMP Database Server System 240a. Here, the SMP Database Server System supports multiple database engines (i.e., multiple instances of the engine 260), including engines 260a through 260n, up to an arbitrary number of database engines.

Detailed Description Text (34):
Ideally, as more processors are added to the SMP server 230a, the Database Server 240a can scale up with additional database engines. Note, however, that certain resources must be shared among the multiple engines, such as global resources require for accessing a common database table. Shared data structures or resource 290, shown in FIG. 2B, is one such resource. So that each engine "sees" the same resource (i.e., has a consistent view), access is protected by a "spin" lock--a well known SMP synchronization mechanism employed for controlling access to lower-level, system critical resources. Note particularly that, as more engines are added to the system, the contention for the shared resource 290 increases. This presents a single bottleneck for all engines.

Detailed Description Text (35):
The shared global structures of the Lock Manager 274 is shown in particular detail in FIG. 2C as shared resource 290. These shared structures, which represent the locking status among multiple users running on multiple engines, are preferred over per-engine data structures. In particular, providing multiple instances of lock management data structures would require less-efficient interprocess communication management--an approach to be avoided. Instead, therefore, the preferred approach is a single set of data structures which are shared among the multiple engine instances.

Detailed Description Text (37):
Given the multi-processing nature of the system of the present invention, access to the hash tables must occur with SMP synchronization. Suppose, for example, that a user task from engine 260a--task.sub.1 --desires to acquire a page lock on the object 294. A page lock is represented by page hash table 293. The task must index into the hash table, for bucket #3. There, the task determines whether the task can be granted the lock or the task must wait for the lock. During this lock operation, task, protects the Lock Manager's structures 290 from change by another task, by tasking out a spin lock. After task.sub.1 has completed the lock operation, it releases the spin lock.

Detailed Description Text (38):
Prior to modification in accordance with the present invention, the Lock Manager's shared data structures 290 were protected by a single spin lock (LOCK.sub.-- SPIN), which serialized access to the address, page, and table hash tables. Whenever a task held this spin lock, it protected the shared Lock Manager data structures entirely (against concurrent changes by other tasks). This approach, to a large extent, simplifies system design. At the same time, however, the approach leads to high contention among multiple engines for the single spin lock. For the example above, for instance, while task, was processing a page lock, another task, say task.sub.2, cannot process a request for an address lock.

Detailed Description Text (42):
Given the multi-processing nature of the system of the present invention, access to the hash tables occurs using the multi-processor synchronization spin lock technique previously described for FIG. 2C. To improve system scalability, however, the spin lock technique is modified. Specifically, the single spin lock (LOCK.sub.-- SPIN) employed for protecting the Lock Manager's hash table data structures is decomposed into a configurable number of spin locks, employed in conjunction with other spin locks as follows:

Detailed Description Text (51):
FIG. 3A illustrates the modification necessary for implementing this design, with particular emphasis on management of address, page, and table locks. In general, each hash table has one or more spin locks, each spin lock for controlling a subset or range of one or more hash buckets for the table. In the exemplary embodiment depicted in FIG. 3A, for instance, the 1031 hash buckets of address lock hash table 302 are controlled by a plurality of spin locks 312 (numbering 10, by default). In a similar fashion, the 101 hash buckets of table lock hash table 304 are controlled by a plurality of spin locks 314 (numbering 5, by default), and the 1031 hash buckets of page lock hash table 306 are controlled by a plurality of spin locks 316 (numbering 10, by default). Thus, for example, a first one of the page spin locks 316 controls access to hash buckets 1-104 for the page lock hash table 306. In this manner, contention is substantially reduced: locks only block when two tasks each concurrently require access to hash buckets covered by the same spin lock. In previous systems, in contrast, a single spin lock was always used to protect such structures and, thus, always presented a bottleneck.

Detailed Description Text (52):
By associating groups of hash buckets with a particular spin lock, spin locks in the system of the present invention are conserved. As described below, the number of spin locks employed for each hash table is user (administrator) configurable. The simplest use is to configure the system such that the minimum number of spin locks is employed: one spin lock for each hash table. Even at this minimum, contention is reduced: processing a lock request of one type of lock (e.g., page lock) will not block access to other types of locks (e.g., address locks). At the other extreme, the user can allocate one spin lock for each and every bucket (up to the number of spin locks available in the system). Here, increasing the number of spin locks increases parallelism available with a given lock type--contention for other hash buckets within a given table decreases as the number of spin locks assigned to that table increases. In typical use, the user will configure or tune use of available spin locks to optimize system performance; the default settings shown in FIG. 3A provide good results for an embodiment scaling up to ten processors. In this manner, contention for access to the Lock Manager can be reduced while preserving efficient use of system spin locks, thereby allowing the system to appropriately scale up as more engines are added to the system.

Detailed Description Text (59):
The Lock Manager processes requests for any type of lock generally as follows. Initially, the Lock Manager receives a task's request for a lock from Access Methods 270. In response to this request, the Lock Manager goes to the hash table associated with the type of lock requested and hashes into the location specified as the address of the object for which the lock is sought; as described above, the Manager takes out the spin lock associated with the bucket being indexed into. At that bucket, the Lock Manager can determine whether any lock objects are present at that

location. If no lock object is found at the indexed-into hash bucket, the desired lock is immediately available and can be provided to the requesting task. If, on the other hand, one or more other tasks have previously requested a lock on the item at that location, the Lock Manager must queue the present request.

Detailed Description Text (61):
2. Exclusive lock processing

Detailed Description Text (62):
The process is perhaps best illustrated by way of example, using an address lock (an exclusive lock). It should be noted, however, that the Lock Manager handles lock requests for other lock types (and other lock strengths, e.g., "intent" locks) in an analogous manner. As shown in FIG. 4A, a hash bucket provides a lock object 410 for address 1. Such a lock object is generated upon a first request for a lock. Here, a request for address lock 1 comes to the Lock Manager from a first task. The Lock Manager evaluates the hash function to identify the appropriate hash bucket for address lock 1, grabbing the spin lock for the group of hash buckets containing that hash bucket. Now, the Lock Manager can traverse the overflow chain in that hash bucket, looking for any lock objects. If the Lock Manager finds no lock object for the address lock 1 on the overflow chain, the Lock Manager automatically creates lock object 410 for address lock 1 and queues that object to the overflow chain.

Detailed Description Text (63):
After the lock object is created, the Lock manager creates a "lock request" 411 for the first task and queues that lock request to a semawait (described below) which is connected to the lock object, as shown in FIG. 4A. The lock record, in this example, is a data structure indicating the task's request for an exclusive lock and includes various pieces of information about the task including the task's process ID. After the lock request 411 has been created and queued, the Lock Manager determines whether that lock record is at the head of the queue off of lock object 410. Because it is, in this example, the requesting task is granted the lock on address 1. The spin lock on the hash bucket is then released.

Detailed Description Text (64):
Next suppose that another request for access to the address 1 lock arrives. This scenario is illustrated in FIG. 4B. The Lock Manager again evaluates the appropriate hash function, grabs a spin lock for the identified hash bucket, and traverses the structures provided in that hash bucket. In this scenario, the Lock Manager finds lock object 410 associated with address lock 1032. Because a lock object currently exists for address lock 1 (i.e., object 410), the Lock Manager need not create a new lock object. The Lock Manager does, however, create a new lock request 413 and appends it to a queue of lock requests hanging off lock object 410. The Lock Manager then determines whether the newly-created lock request 413 is at the head of the queue. In the example of FIG. 4B, it is not at the head as the task associated with lock request 411 currently holds an exclusive lock on the address lock. Therefore, the second task must wait for the lock; at this point, it can go to sleep.

Detailed Description Text (67):
3. Shared lock processing

Detailed Description Text (68):
The foregoing example illustrates an address lock--an exclusive lock. For an exclusive lock, if the request is at the head of the queue, the request is granted, as no two exclusive lock requests can simultaneously reside at the head of queue. Page and table locks, in contrast, may be either exclusive or shared (non-exclusive). As noted, a non-exclusive lock may permit one or more other non-exclusive locks to be taken out on the same object (e.g., database page). To implement non-exclusive or shared locks, the Lock Manager chains another data structure, a semawait, in addition to the above-described lock object and lock request data structures. The semawait indicates the position in the queue; each position has one or more shared lock requests or a single exclusive lock request. For efficiency in system design, all of these locking data structures share the same size and are allocated from the freelock list (described in detail below).

Detailed Description Text (69):

One exemplary process for generating an overflow chain for page locks will now be described with reference to FIG. 4C. Initially, a first task requests a non-exclusive lock on a page 2. In response, the Lock Manager identifies the appropriate hash bucket for the page 2 lock, takes out a spin lock on the associated subset of hash buckets, and hashes into the identified hash bucket. Assuming that the page 2 lock is not currently taken, Lock Manager 274 discovers that no lock object exists for page 2 and creates lock object 420. Because the page lock request is for a non-exclusive lock, Lock Manager 274 then creates a semawait 430 which it appends directly off of page object 420, at the head of queue. Note that semawaits are employed only for non-exclusive locks. After the semawait structure has been created and queued, the Lock Manager creates a first non-exclusive lock request 431, appended directly off of semawait 430, and containing information about the first task. The first task then releases the spin lock on the subset of hash buckets.

Detailed Description Text (70):
Next, a second task also requests a non-exclusive lock on page 2. At this point, the Lock Manager grabs the appropriate spin lock, hashes into the hash bucket, and discovers page lock object 420 with semawait 430 and non-exclusive lock request 431. The Lock Manager then produces a second non-exclusive lock request 433, which it queue up (i.e., appends directly to the prior non-exclusive lock record 431). Both the first and second tasks now hold non-exclusive locks on page 2 and continue executing with shared access to page 2.

Detailed Description Text (71):
Now, consider a third task which requests an exclusive lock on page 2. This scenario is illustrated in FIG. 4D. Because this exclusive lock is incompatible with the non-exclusive locks currently held on page 2, the third task's request cannot be granted immediately. Thus, Lock Manager 274 generates a semawait 440 with a lock request 441 for the third task and queues it to the main overflow chain off the lock object for page 2. At this point, the third task will wait for the prior semawait to be dequeued; at this point, it can go to sleep.

Detailed Description Text (72):
Next, consider a fourth task which also requests an exclusive lock of page 2. Again, this request cannot be granted because the desired exclusive lock would not be lock compatible with the currently held non-exclusive locks. Further, note that this exclusive lock would not be lock compatible with the prior exclusive lock, as exclusive locks cannot co-exist on the same object. Thus, Lock Manager 274 generates yet another semawait 450 and appends it to the overflow chain with a lock request 451 for another exclusive lock, as illustrated in FIG. 4E. The fourth task now waits on this newly-added lock request; it can sleep at this point.

Detailed Description Text (73):
Finally, consider a fifth task which requests a non-exclusive lock on page 2. This non-exclusive lock is lock compatible with the currently held non-exclusive locks of the first and second tasks. As shown in FIG. 4F, therefore, the Lock Manager links the corresponding lock request 435 to the first semawait 430, thereby granting the newly-requested non-exclusive lock to the fifth task.

Detailed Description Text (74):
After the first, second, and fifth tasks deallocate their non-exclusive locks, semawait 430 and its non-exclusive lock requests are dequeued. As a result, the second semawait 440 and its exclusive lock request 441 move to the head of the queue. Concurrently therewith, the third task wakes up and continues execution with an exclusive lock on page 2. When this task subsequently deallocates its exclusive lock structures, the third semawait 450 and its exclusive lock request 451 move to the head of the queue. The fourth task thus continues execution with its requested exclusive lock on page 2. Thereafter, the task dequeues the lock request. After all lock requests and semawaits have been dequeued, the lock object itself is de-allocated.

Detailed Description Text (75):
Note that the above scenario for handling requests for both exclusive and non-exclusive locks could allow the third and fourth tasks to remain waiting indefinitely while additional requests for non-exclusive locks are received and

queued off of the first semawait 430. To address the situation, the Lock Manager is preferably designed such that a queued request for an exclusive lock waits on no more than a certain number of non-exclusive lock requests lodged after it was queued (i.e., "skip" factor). In one particular preferred embodiment, a task requesting an exclusive lock will allow no more than four non-exclusive locks requests to be queued before it, after it has made its request for the exclusive lock. In this case, if an fifth request for a non-exclusive lock comes in, a new semawait is created and appended to the end of the overflow chain (behind any lock requests for exclusive locks). In this manner, subsequent requests for non-exclusive locks are queued through a subsequent semawait, rather than the semawait in front of the exclusive lock record.

Detailed Description Text (87):
Because the Lock Manager employs multiple spin locks when traversing Lock Manager data structures (namely, the hash buckets and their chains), deadlock detection should account for changes in lock status during the course of the search. Note that the various spin locks on the hash buckets are not frozen during the search so locks may be released and/or granted after the search is started but before it ends. Thus, deadlock searching itself is are dynamic--occurring while the system itself continues to operate. As a result, care must be taken to ensure that the data structures relied upon remain consistent. In contrast, if a single spin lock is provided for the entire Lock Manager, as was the case in with prior art systems, no data structures within the Lock Manager can change during the course of the deadlock search. That approach is simpler, but systems implementing that approach are effectively frozen while deadlock detection is underway.

Detailed Description Text (97):
To reduce the risk of detecting spurious deadlocks, the search algorithm preferably restricts its traversal to only those dependency edges that were already formed when the search started. The search is limited to those edges having "sequence numbers" less than or equal to the sequence number in existence for the task when the search began. Each number is an incrementally-increasing number assigned to a task which has to wait for a lock. During the search, the system uses the lock wait sequence number on the lock sleeptask queue to detect and discard edges that were formed after the search started. When performing deadlock detection, any edge the task is traversing which has a sequence number which is less than or equal that for the task will be examined. If the sequence number is greater than that for task, a new edge has formed and, thus, deadlock search need not be continued. Any new cycles formed from new edges may be detected by later searches (initiated after the cycles are formed). For the case of FIG. 6C, the search would disregard edge L7 having a sequence number of 3 because that edge was created after the search began. Specifically, when the search begins, the initiating task grabs the spin lock on the sleeptask queue and identifies the sequence number at that time. With this number, it can decide whether to record or ignore edges it encounters in its search.

Detailed Description Text (99):
FIG. 7 illustrates how the deadlock search method treats a cycle in which the initiating task does not part of the cycle. Initially, a task T1 queues a lock request L1 but is blocked because of a shared lock held by two tasks T2 and T5. It detects no deadlock when it traverses the edge (L1) to T5; however, it does identify a cycle when it traverses to task T4 through a task T3. At this point, the dependency graph contains a path in T1, T2, T3, and T4, with a cycle. Note, however, that T1 is not participating in the cycle and, thus, can be ignored. Nevertheless, the detection method does identify this as an "innocent bystander" deadlock. In a preferred embodiment, the search method does not try to break these types of deadlocks because of the expense involved. To choose an optimal set of victims, the method would have to record all the cycles it detects. Then it would have to select victims in a way that breaks all cycles, but leaves the least number of tasks victimized.

Detailed Description Text (105):
The actual locks themselves represent resources in the system represented internally by various underlying lock data structures. Each lock is either "free" (i.e., available for use) or "allocated" (i.e., in use). These data structures are maintained on a global "freelock" list in shared memory and which is available to

all server engines. Lock requests are satisfied, in part, by allocating lock data structures from this list. The approach of a single global freelock list presents a bottleneck--a single point of contention, however. More particularly, many of the performance gains of parallel lock management would remain unrealized if a bottleneck elsewhere in the process remained uncorrected. The present invention addresses this problem by providing improved methodology for global freelist management.

Detailed Description Text (107):
Actual "movement" of locks is logical not physical. Here, the locks are not physically moved from a global list memory location to the engine-specific freelock caches but, instead, remain in the shared memory location allocated for the global freelock list at start up time. Changes are made to the pointers that link the lock structures together, so that when locks are "moved," they are no longer referenced as part of a "free" lock on the global freelock list.

Detailed Description Text (108):
FIG. 9 diagrammatically illustrates the above-described approach. As shown, a global freelock list 900 containing a global freelock list 910. These are available to the Lock Manager of each engine. As shown, for instance, "free" locks in list 910 are available to engine freelock caches such as cache 920 for engine 0, cache 930 for engine 1, and cache 940 for engine 2. Thus, one group of freelocks 921 has been transferred from list 900 to cache 920, another group of freelocks 931 is transferred to cache 930, yet another group of freelocks 940 is transferred to cache 941, and so forth and so on, up to an arbitrary number of engines. In the preferred embodiment, the actual size of each engine freelock cache and therefore the number of "free" locks which each holds is user configurable. With this approach of binding certain freelocks to particular engine caches, less contention exists for the global freelock list 900. Specifically, if a server engine requires a lock, it first looks to its own freelock cache, without tying up the global freelock list (and contending for its single spin lock). Only when no more freelock structures are left in an engine's own cache will a configurable number of freelock structures be moved from the global freelock list to the engine's own freelock cache.

Detailed Description Text (110):
In preferred exemplary embodiments of the present invention, the end user--a database administrator (DBA) or application programmer--can configure the operation of the Lock Manager. The user may effect this by setting certain configuration parameters, for example. Such parameters control, for example, the deadlock search, the spin lock decomposition, and the freelock cache.

Detailed Description Text (115):
Still further, the Lock Manager may be designed such that the end user may specify the size of a "freelock cache." As noted, all locks provided by the Lock Manager are designated as either free or allocated. In this embodiment, the end user may specify what percentage of the total locks may be designated as freelocks for the database engines. It may also specify how many freelocks are transferred from the global freelock list to a server freelock cache when there are no more freelocks left in a server's freelock cache.

Detailed Description Text (123):
The lock.sub.-- psema API call, at lines 17-19, is employed for a special locking purpose: locking the log page (i.e., the last page of the log). The lock.sub.-- psema function serves to hold a lock briefly--releasing the held lock at the end of the log operation. The lock.sub.-- vsema call, at lines 21-22, functions in a similar manner.

Detailed Description Text (130):
The first two lock types, EX.sub.-- TAB and SH.sub.-- TAB, indicate an exclusive table lock and a shared table lock, respectively. The next two lock types, EX.sub.-- INT and SH.sub.-- INT, indicate an exclusive intent lock and a shared intent lock, respectively. An "intent" lock is employed in the context of acquiring page locks. Here, the client employs the "intent" lock to make sure that the entire table is not locked. In other words, the client employs an intent lock on a table when the client is holding page locks in that table. The page-level locks, EX.sub.-- PAGE, SH.sub.--

PAGE, and UP.sub.-- PAGE, represent an exclusive page lock, a shared page lock, and an update page lock, respectively.

Detailed Description Text (132):
As shown, two types of locks are provided: an exclusive address lock (EX.sub.-- ADDR) and a shared address lock (SH.sub.-- ADDR). The shared address lock is provided so that multiple readers of an object, such as multiple readers scanning the pages of an index, can do so in a shared fashion, thereby avoiding the need for the readers to serialize one after another.

Detailed Description Text (142):
At line 72, the routine performs general setup and initialization steps. Then, at line 74, it checks the lock mode, which specifies how a lock is released when traversing from parent to child page. At lines 76-83, the actual page is retrieved in memory, by a call to getpage. At lines 85-91, the Lock Manager is invoked for obtaining an appropriate address lock on the page (either shared or exclusive).

Detailed Description Text (143):
As shown at line 89, the actual call to the Lock Manager is made through a macro, RLOCK.sub.-- EXCLUSIVE, for ladder locking; otherwise, the call to the Lock Manager is made through the RLOCK.sub.-- SHARE macro at line 91 (for timestamp locking). These macros, in turn, expand out into calls to the Lock Manager's lock.sub.-- address API function, as shown by the following macro definitions.

Detailed Description Text (144):
define RLOCK.sub.-- EXCLUSIVE (bp, sdes, lockrecp).backslash.lock.sub.-- address (LOCK.sub.-- ACQUIRE, EX.sub.-- ADDR, (char *) bp, &sdes.fwdarw.slocks, lockrecp)

Detailed Description Text (146):
For a ladder locking traversal of the index pages, therefore, an exclusive lock is requested by the getindexpage function as follows:

Detailed Description Text (148):
As shown, the first parameter to the lock.sub.-- address call specifies a LOCK.sub.-- ACQUIRE operation. The second parameter indicates the type of lock required, here, an exclusive address lock (EX.sub.-- ADDR). The address for which the lock is required, the third parameter, is simply the address in memory (i.e., pointer to the index page; here, bp--a buffer). The fourth parameter, slocks, specifies a context chain which the client provides (via the session descriptor, sdes). Here, the session descriptor or sdes includes a slocks member, for storing the address of the lock context chain (i.e., the linked list of lock records).

Detailed Description Text (149):
The lock.sub.-- address function serves to acquire a lock on a data server address. Typically, the address will correspond to a data structure associated with an object, such as a buffer header. Address locks can be requested in shared or exclusive lock types. Compatibility for these lock types is represented by the following table.

Detailed Description Text (150):
As shown, the only compatibility which exists is between two shared address locks. Address locks are granted in the order that they are requested, thereby avoiding "starvation" of exclusive address requests. If there exists multiple readers, for instance, no reader bypasses an exclusive access. This approach leads to increased efficiency of the code (which itself is executed frequently).

Detailed Description Text (152):
The steps of the function are as follows. Local (stack) variables are declared initially, at lines 80-86. At lines 89-96, the function hashes the passed-in address into an address lock hash table. This yields a particular "bucket." Here, the bucket is protected by a spinlock; at line 96, this spinlock is grabbed. At lines 98-106, the function checks to make sure that the current task has enough locks cached. If not, more locks will be obtained from a shared/global freelist of locks.

Detailed Description Text (153):

At lines 106-114, the spinlock is asserted, thereby assuring exclusive access to the hash bucket (for guaranteeing contents and integrity of the data structures). The spinlock will be held until termination of the routine. At lines 116-125, the function looks for the object(s) which hang off the hash bucket (i.e., hash overflow chain). If the address (for the address lock) already exists in the hash bucket, an address lock has already been taken out for that object. If the address is not found there, then the current request is the first lock request for the bucket, for that particular address. At lines 127-142, a lock object is allocated (if one does not already exist in the hash table) and is linked into the hash chain. This provides a head (semaphore) in which "waiters" (i.e., those waiting for lock requests to be granted) can be chained from. The locked object is linked to the hash table using a macro, as shown specifically at line 141.

Detailed Description Text (155):
(1) Request is for an exclusive address lock;

Detailed Description Text (156):
(2) Request is for a shared address lock but the semawait queue for the lock object is empty; or

Detailed Description Text (157):
(3) Request is for a shared address lock but the last semawait in the semawait queue for the lock object is for an exclusive address lock.

Detailed Description Text (159):
An exclusive address lock has the property that there will exist only one lock record per semawait. In other words, a semawait will be allocated and an address lock will be allocated. If, on the other hand, the lock type is shared, multiple lock records or objects can be hanging off one semawait. Lines 161-163 specifically test if the locktype is an "exclusive address" type, or there exists no SEMAWAIT in the object (i.e., the object is linked to itself--no other SEMAWAIT is hanging off), or the head is equal to EX.sub.-- ADDR. If any of these conditions hold true, then a new SEMAWAIT is allocated (line 165) and linked in (line 168). Otherwise, the function uses the tail semawait. Here, the tail semawait already has a shared address lock and the incoming lock is a shared address lock, which can be queued to the tail.

Detailed Description Text (168):
The deadlock search method employs a global sequence number and a queue. The global sequence number is called a "lock wait sequence number." This sequence number is initialized at server start-up time and incremented whenever a task has to wait for a lock. The sequence number is incremented under LOCK.sub.-- SLEEPTASK.sub.-- SPIN spinlock by the task which has to wait for its lock request. The queue, on the other hand, is called a "lock.sub.-- sleeptask queue." Each task which has to wait for a lock adds an entry to lock.sub.-- sleeptask.sub.-- queue. Once the lock is granted, the task will remove its entry from this queue. The queue comprises one or more "sleeptask" records. Each record describes the sleeping task. It stores the semawait on which the task is sleeping along with the lock wait sequence number at the time it went to sleep. Lock wait sequence number will change when the task initiates a deadlock search.

Detailed Description Text (172):
Since the deadlock search holds only the spinlock corresponding to one hash bucket at a time, lock operations can proceed on other hash buckets. In other words, the method allows the dependency graph to change while the deadlock search is in progress. As a result, new dependency edges might form and old edges might be deleted as the search is traversing the dependency graph. For correctness of the search, the search method restricts the traversal to dependency edges that were formed since the search started. Any cycles formed because of new edges are detected by later searches initiated by the tasks which created the edges. The sequence number on the lock.sub.-- sleeptask.sub.-- queue is used to detect and discard edges that were formed after the search started. Any entry added to the queue after the search started has a sequence number greater than the sequence number when the deadlock search started.

Detailed Description Text (207):
While the invention is described in some detail with specific reference to a single preferred embodiment and certain alternatives, there is no intent to limit the invention to that particular embodiment or those specific alternatives. For example, while locks and many data structures associated with locks have been described as accessible through hash tables, those skilled in the art, enabled by the teachings of the present invention, will appreciate that other efficient-indexing access mechanisms may be employed. Further, although the discussion of the preferred embodiment focuses on exclusive and non-exclusive locks, those skilled in the art will appreciate that the teachings of the present invention may be applied to other types of locks (e.g., "intent" locks), such as the varying degrees of lock exclusivity and granularity commonly provided by database systems. In such a case, compatibility between locks can be determined by providing a lock compatibility table. Thus, the true scope of the present invention is not limited to any one of the foregoing exemplary embodiments but is instead defined by the appended claims.

Detailed Description Paragraph Table (5):

```
/* ** LOCKREC ** ** A lockrec identifies a task waiting for a semaphore, and facilitates ** backout by being linked onto the task's global data structures. ** ** NOTE: lockobjs, semawaits, lockrecs, and freelocks are all the ** same size so they can be freed to the same free list. */ struct lockrec /* lrprnext and lrprprev must be first to match up with lockhead */ struct lockrec *lrprprev; /* prev lock on sdes, xdes, or pss */ struct lockrec *lrprnext; /* next lock on sdes, xdes, or pss */ struct spinlock *lrspinlock; /* spinlock for this hash bucket */ /* lrnext and lrprev must be next to match up with semawait */ struct lockrec *lrprev; /* prev lock on semawait */ struct lockrec *lrnext; /* next lock on semawait */ spid.sub.-- t lrspid; /* spid of task holding or waiting */ /* spid = server process ID */ BYTE lrtype; /* lock type for logical locks */ BYTE lrstatus; /* lock flags */ int16 lrsuffclass; /* sufficiency class */ 0x6c72 */rmagic; /* "lr" struct semawait *lrsemawait; /* semawait for these locks */ };
```

Detailed Description Paragraph Table (7):

```
__ 1: /* 2: ** GETINDEXPAGE 3: ** 4: ** Traverse an index from parent to the child page given in 5: ** sdes-->scur. Return a buffer pointer to the kept, rlocked child. 6: ** This routine is the only way to safely traverse from parent to 7: ** child in an index. It has a symbiotic relationship with its only 8: ** caller, srchindex. Getindexpage implements two searching methods 9: ** based on the ladderlock flag. 10: ** 11: ** If `ladderlock` is TRUE, hold the rlock on the parent while 12: ** reading and rlocking the child. This guarantees the identity of 13: ** the child (because the parent must be updated to split or shink 14: ** the child) , and ensures that progress will be made, but reduces 15: ** index concurrency. With ladder locking, both parent and child 16: ** pages are kept and rlocked on return. 17: ** If `ladderlock` is FALSE, drop the rlock on the parent before 18: ** rlocking the child, and verify the child's identity by 19: ** rechecking the parent's timestamp (the parent would have to have 20: ** changed for the child's identity to have changed). This scheme 21: ** increases index concurrency because only one rlock is held at a 22: ** time. However, in an index with many splits and/or shrinks, a 23: ** search using this method may never make progress. With this 24: ** method the child is returned kept and rlocked while parent is 25: ** unrlocked and unkept. 26: ** For either case of timestamp locking or ladder locking, if 27: ** a deadlock is detected then we will tolerate this deadlock by: 28: ** 29: ** 1. If ladder locking then release the parent rlock and wait 30: ** on the child rlock. When get the child rlock, release 31: ** it and return failure to start index search over again. 32: ** 33: ** 2. If timestamp locking then parent has already been released 34: ** so simple wait on the child and when get it release it and 35: ** return failure and start the index search over again. 36: ** 37: ** We wait on the child page and immediately release it to prevent 38: ** thrashing on the index structure. 39: ** 40: ** PARAMETERS: 41: ** srchlock.sub.-- info - Index lock structure holding relevant lock 42: ** info 43: ** RETURNS 44: ** Pointer to kept buffer for child page, or NULL if the desired 45: ** page could not be obtained for some reason. If NULL is 46: ** returned, then sdes-->sridoff holds: 47: ** Child may have changed identity; 48: ** caller can restart search. 49: ** Deadlock obtaining child rlock. 50: ** 51: ** SIDE EFFECTS 52: ** If ladder locking, both parent and child are kept
```

and rlocked 53: ** on return. If not ladder locking, parent is unrlocked and 54: ** unkept, and child is rlocked and kept. On error, both parent 55: ** and child are unrlocked and unkept. 56: ** Ex.sub.-- raise if a page belonging to the wrong object is arrived at. 57: ** Ex.sub.-- raise if attention condition raised. 58: ** 59: ** MP SYNCHRONIZATION: 60: ** The search methods rely on the way that indexes are changed in 61: ** order to guarantee that the search is not misdirected to the 62: ** wrong page. Both methods rely on rlocks being obtained on all 63: ** pages involved in a split or shrink and being held until the 64: ** split or shrink is complete. The timestamp method requires that 65: ** the page timestamp changes be made while the rlocks are held. 66: ** 67: */ 68: BUF * 69: getindexpage(INDEXLOCK * srchlock.sub.-- info) 70: { 71: 72: /* . . . do setup and initialization */ 73: 74: /* . . . check the lock mode (timestamp or ladder locking) */ 75: 76: /* get the page in memory */ 77: /* 78: ** We do not want a DEADLOCK error to make it back to the client 79: ** here as we will retry the RLOCK( ) below. Update bp in 80: ** srchlock.sub.-- info, so we can cleanup our resources held. 81: */ 82: bp = getpage(sdes); 83: *srchlock.sub.-- info-->bp = bp; 84: /* 85: acquire the appropriate address lock on the page 86: (shared or exclusive); 87: */ 88: if (ladderlock) 89: lockstat = RLOCK.sub.-- EXCLUSIVE(bp, sdes, srchlock.sub.-- info-->bpl ock); 90: else 91: lockstat = RLOCK.sub.-- SHARE(bp, sdes, srchlock.sub.-- info-->bplock) ; 92: 93: /* . . . */ 94: }

Detailed Description Paragraph Table (9):

1: /* 2: ** LOCK.sub.-- ADDRESS 3: ** 4: ** Acquire a lock ona dataserver address. 5: ** Generally the address will correspond to a 6: ** datastructure associated with an object, such as 7: ** a buffer header. 8: ** Address locks available in shared or exclusive lock types. 9: ** 10: ** Compatibility table for these two lock types. 11: ** 12: ** EX.sub.-- ADDR SH.sub.-- ADDR 13: ** 14: ** 15: ** 16: ** 17: ** ##STR1## 18: ** 19: ** Address locks are granted in the order that they are 20: ** requested. This avoids starvation of EX.sub.-- ADDR requests. 21: ** 22: ** Parameters: 23: ** 24: ** op Operation to perform. The following operations are 25: ** supported: 26: ** 27: ** LOCK.sub.-- ACQUIRE Acquire a lock on the given address. 28: ** 29: ** The following value can be or'd with LOCK.sub.-- ACQUIRE: 30: ** 31: ** LOCK.sub.-- NOWAIT Do not sleep if the lock cannot be 32: ** granted immediately. The caller must then call 33: ** lock.sub.-- wait to wait for it. 34: ** 35: ** locktype The type of this address lock request. 36: ** EX.sub.-- ADDR or SH.sub.-- ADDR. 37: ** 38: ** address An address to lock. Subsequent lockers of this 39: ** address will have to wait. 40: ** 41: ** ctxchain The context chain to which to link the lock. 42: ** 43: ** lockrecp A pointer to a pointer to a lockrec, or NULL. If 44: ** it is non-null, lock.sub.-- address will fill it in with a 45: ** pointer to the lock of interest. This may then *only* be 46: ** passed to lock.sub.-- wait or lock.sub.-- release. 47: ** 48: ** Returns: 49: ** 50: ** One of these status values is returned: 51: ** 52: ** LOCK.sub.-- GRANTED (+) The lock was granted immediately. 53: ** LOCK.sub.-- WAITED (+) The caller's task had to wait for the lock 54: ** to be granted. The lock WAS granted. 55: ** LOCK.sub.-- DIDNTWAIT (-) The caller would have had to wait, but 56: ** LOCK.sub.-- NOWAIT was specified. Lockrecp was filled 57: ** in with the lockrec to wait for later. 58: ** LOCK.sub.-- DEADLOCK (-) The caller was selected as a deadlock victim. 59: ** LOCK.sub.-- INTERRUPTED (-) The lock was not granted because the task 60: ** received an attention condition while 61: ** waiting for it. 62: ** 63: ** MP Synchronization: 64: ** No explicit synchronization required of the caller. Obtains 65: ** the spinlock from lock address hash bucket to protect lock 66: ** states and chains. 67: ** 68: ** Side Effects: 69: ** Before we ex.sub.-- raise an error in this routine we will turn off 70: ** P2.sub.-- LOCKRETRY status bit in the pss. We don't have to do the 71: ** same for each return as the caller of this routine with this 72: ** status bit set will clear out this bit in the pss. 73: ** This status bit should only be used by srchindex(). 74: ** 75: */ 76: int 77: lock.sub.-- address(int op, int locktype, char * address, LOCKHEAD * 78: ctxchain, LOCKREC ** lockrecp) 79: { 80: LOCALPSS(pss); /* initialize local copy of global pss */ 81: LOCKOBJ *bucket; /* hash table overflow chain pointer */ 82: LOCKOBJ *obj; /* object we're locking */ 83: SEMAWAIT *sw; /* semawait cursor */ 84: LOCKREC *lr; /* lockrec cursor */ 85: int ret; /* return value */ 86: SPINLOCK *spinlock; /* spinlock for the hash chain */ 87: 88: 89: /* 90: ** Hash the address into the address lock hash table. This can 91: ** be done without the spinlock since we are simply doing an 92: ** array index.

93: */ 94: 95: bucket = LOCK.sub.-- .sub.-- ADDR.sub.-- HASH(address); 96: spinlock = ((HASHLOCKHEAD *) bucket)->hlhspinlock; 97: 98: /* 99: ** Before getting the spinlock for the hash chain, make sure 100: ** there this task has enough locks cached. If not, get some 101: ** more from the shared free list. 102: */ 103: if (Eresource->erfreelock.sub.-- cache.fcfreelocknum < MINLOCKSTRUCTS) 104: { 105: lock.sub.-- .sub.-- fillcache(); 106: } 107: 108: /* 109: ** Get hash bucket spinlock in order to search chains and 110: ** guarantee contents and identity of the data structures. It 111: ** won't be released until this routine returns. 112: */ 113: 114: P.sub.-- SPINLOCK(spinlock); 115: 116: /* 117: ** Look for the object on the hash overflow chain. 118: */ 119: 120: for (obj = (LOCKOBJ *) bucket->lonext; obj != (LOCKOBJ *) bucket; 121: obj = obj->lonext) 122: { 123: if (obj->loobjidpageno == (long) address) 124: break; 125: } 126: 127: /* 128: ** Allocate a lockobj if there wasn't one in the hash 129: ** table already, and link it onto the hash chain. 130: ** This provides a head (semaphore) to chain waiters from. 131: */ 132: if (obj == (LOCKOBJ *) bucket) 133: { 134: /* Allocate and link onto hash table. */ 135: LOCK.sub.-- .sub.-- ALLOC.sub.-- LOCKOBJ(Eresource->erfreelock.sub.-- cache, obj); 136: 137: /* Identify the object. */ 138: obj->loobjidpageno = (long) address; 139: 140: /* Link to the hash table */ 141: LINK TO HASHTABLE(bucket, obj, spinlock); 142: } 143: 144: /* 145: ** Allocate a semawait, initialize it, and link it to 146: ** the tail of the semaphore queue. 147: ** Allocation of the new semawait will be done when any one of 148: ** the following conditions is true: 149: ** 150: ** 1) Request is for an exclusive address lock. 151: ** 2) Request is for a shared address lock but the semawait queue for 152: ** the lock object is empty. 153: ** 3) Request is for a shared address lock but the last semawait in 154:

Detailed Description Paragraph Table (10):

** the semawait queue for the lock object is for an exclusive 155: ** address lock. 156: ** 157: ** Note that condition 3) maintainss the FIFO order of address 158: ** lock requests and avoids starvation. 159: ** 160: */ 161: if ((locktype == EX.sub.-- ADDR) .vertline..vertline. 162: ((obj->losemaphore.smtail == (SEMAWAIT *)obj) .vertline..vertline. 163: (obj->losemaphore.smtail->swlhead->lrtype == EX.sub.-- ADDR))) 164: { 165: LOCK.sub.-- .sub.-- ALLOC.sub.-- SEMAWAIT(Eresource->erfreelock cache, sw); 166: sw->swstatus = 0; 167: sw->swspinlock = spinlock; 168: LINK.sub.-- TO SEMAPHORE(&obj->losemaphore, sw); 169: } 170: else 171: { 172: /* 173: ** This is a shared lock request and one of the following 174: ** conditions is true: 175: ** 1) The last waiter is also a shared lock request. 176: ** 2) All the current owners are shared mode owners and there 177: ** is no waiter. Note that there would only be one semawait 178: ** in the queue in this case. 179: ** In either case we do not have to allocate a new semawait. We 180: ** can just queue the current request to the last semawait in 181: ** the queue. 182: */ 183: sw = obj->losemaphore.smtail; 184: } 185: /* Allocate a lockrec, initialize it, 186: and link it to the semawait. */ 187: LOCK.sub.-- .sub.-- ALLOC.sub.-- LOCKREC(Eresource->erfreelock.sub.-- cache, lr); 188: lr->lrspid = pss->pspid; 189: lr->lrstatus = 0; 190: lr->lrtype = locktype; 191: lr->lrsuffclass = LOCKSUFFCLASS.sub.-- NONE; 192: LINK.sub.-- TO.sub.-- SEMAWAIT(sw, lr, spinlock); 193: 194: /* Sdes's sometimes get cleared. This would make the context chain 195: ** invalid, since it's supposed to point to itself when empty. 196: ** If we detect a cleared-out context chain, presume it's tneant to 197: ** be empty and re-initialize it. Obviously it would be better 198: ** to fix sdes management, but that's not too easy. 199: */ 200: if (ctxchain->lhhead == 0) 201: MKPSSLKHD((*ctxchain)); 202: 203: /* Link the lock to the context chain */ 204: LINK TO.sub.-- LOCKHEAD(ctxchain,.lr); 205: 206: /* 207: ** The lock has been created and found it's place in line, 208: ** now see what to do about it. 209: */ 210: 211: /* Lock is at the head of the queue; grant it immediately. */ 212: if (sw == obj->losemaphore.smhead) 213: { 214: if (lockrecp) 215: *lockrecp = lr; 216: lr->lrstatus .vertline.= LR.sub.-- GRANTED; 217: V.sub.-- SPINLOCK(spinlock); 218: return(LOCK.sub.-- GRANTED); 219: } 220: 221: /* Caller has to wait but doesn't want to, at least now. */ 222: if (op & LOCK.sub.-- NOWAIT) 223: { 224: if (lockrecp) 225: *lockrecp = lr; 226: /* Mark the lockrec so we can tell that even though 227: ** the semawait is not at the head of the queue, this task 228: ** still won't be waiting for it (until it calls lock wait) 229: */ 230: lr->lrstatus .vertline.= LR.sub.-- NOTWAITING; 231: V.sub.-- SPINLOCK(spinlock); 232: return(LOCK.sub.-- DIDNTWAIT); 233: } 234: 235: /* Otherwise, wait for the lock. Lock.sub.-- semawait 236: ** releases the spinlock. 237: */ 238: ret = lock.sub.-- semawait(sw, lr); 239: 240: if (lockrecp) 241: *lockrecp = (ret ==

LOCK.sub.-- WAITED ? lr : 0); 242: 243: return (ret); 244: }

Detailed Description Paragraph Table (12):

179: V.sub.-- SPINLOCK(RDATETIME.sub.-- SPIN); 180: 181: /* 182: ** Sleep for the semaphore. It is not necessary to obtain the 183: ** hash bucket spinlock again since there is one wakeup per 184: ** semawait; when this task wakes up, our caller will obtain 185: ** the hash bucket spinlock if necessary. 186: ** However, there could be a late wakeup from a previous use 187: ** of this semawait. So we have to sleep until the pwaitsema 188: ** field in the pss is cleared. 189: */ 190: sleepfor.sub.-- lock: 191: 192: /* Increase this task's priority. This is because once 193: ** the lock is released, the first task on the queue owns 194: ** it, even while it's asleep. Increasing the priority 195: ** here means the task will run sooner once it's granted 196: ** the lock. Running sooner hopefully means it will 197: ** release the lock sooner, decreasing lock contention. 198: */ 199: (vold) uppri(-1); 200: 201: /* Wait for the lock. To avold race conditions with both 202: ** normal lock releases and deadlock wakeups via uppwakeup, 203: ** the PL.sub.-- SLEEP bit is set in the plockstat field instead of using 204: ** the SEMA.sub.-- WAITING bit in the semawait. 205: */ 206: (void) upsleepgeneric(SYB.sub.-- EVENT.sub.-- STRUCT(semawait), 207: (char *) &pss-->plockstat, sizeof(pss-->plockstat) 208: (long) PL.sub.-- SLEEP, 1); 209: 210: /* Restore priority now that we've woken up. */ 211: (vold) uppri(1); 212: 213: 214: /* 215: ** Check to see if lockrec was moved to another semawait 216: ** while we were sleeping. This could happen if lock.sub.-- regueue( ) 217: ** was called. For details see the prologue of 218: ** lock.sub.-- requeue( ) and the code section where we set the bit. 219: */ 220: P.sub.-- SPINLOCK(spinlock); 221: if (lockrec-->lrstatus & LR.sub.-- MOVED) 222: semawait = lockrec-->lrsemawait; 223: 224: /* 225: ** There are four reasons for us to wake up from 226: ** sleep: 227: ** 1. We got the lock! In this case we get out in a hurry. 228: ** 2. We were woken up because of an attention. 229: ** 3. We were woken up to initiate deadlock search. In this 230: ** case we have to initiate deadlock search. 231: ** 4. We were selected as a deadlock victim. In this case we 232: ** need to cleanup state and return with LOCK.sub.-- DEADLOCK status. 233: ** We will handle each of these conditions in order. 234: */ 235: 236: /* We got the lock. */ 237: if (lockrec-->lrstatus & LR.sub.-- GRANTED) 238: { 239: /* We can release the spinlock now. */ 240: V.sub.-- SPINLOCK(spinlock); 241: /* 242: ** At this point this task should own the semaphore, and the 243: ** semawait should be at the head of the queue. If this is 244: ** true, neither the swstatus field nor the semaphore head 245: ** pointer can change, even though this task doesn't hold the 246: ** hash bucket spinlock. This sanity test is the bottom line 247: ** for correct operation of the lock manager so it's left in 248: ** even if SANITY isn't defined. 249: */ 250: if ((semawait-->swstatus & SEMA.sub.-- WAITING) 251: .vertline. .vertline. semawait-->swsemaphore-->smhead != semawait) 252: { 253: ex raise(LOCKM, LKPREMATUREWAKE, EX.sub.-- CMDFATAL, 1); 254: } 255: 256: goto gotlock; 257: } 258: 259: /* 260: ** See if our sleep may have been interrupted. 261: */ 262: if ((PSS.sub.-- GOT.sub.-- ATTENTION(pss)) .vertline. .vertline. (pss-->pstat & P.sub.-- KILLYOURSELF)) 263: { 264: /* 265: ** If we are here, it means that the lock has not been granted 266: ** yet. This is because we checked whether the lock was 267: ** granted before checking for attention and we have not yet 268: ** released the spinlock. All that remains to be done is to 269: ** cleanup and return with LOCK.sub.-- INTERRUPTED status. 270: */ 271: 272: /* Release lockrec and mark the task as not waiting for lock. */ 273: pss-->pwaitsema = 0; 274: event = lock.sub.-- unlink(lockrec); 275: V.sub.-- SPINLOCK(spinlock); 276: 277: if (event) 278: (vold) upwakeup(SYB.sub.-- EVENT.sub.-- STRUCT(event)); 279: 280: /* Drain the freelock cache if necessary */ 281: CHECKFREELOCKCACHE.sub.-- OVERFLOW (Eresource-->erfreelock.sub.-- cache, 282: Resource-->rcmaxfreelock.sub.-- engine); 283: 284: /* Remove lock sleep queue entry */ 285: P.sub.-- SPINLOCK(LOCK.sub.-- SLEEPTASK.sub.-- SPIN); 286: LOCK.sub.-- DELETE.sub.-- SLEEPTASK(pss, ststatus); 287: /* 288: ** Pick the next task to perform deadlock search if we were to 289: ** have initiated a deadlock search now. 290: */ 291: if (ststatus & STCHECKDEADLOCK) 292: { 293: /* This routine will release LOCK.sub.-- SLEEPTASK.sub.-- SPIN */ 294: lock.sub.-- check.sub.-- timeout(LOCK.sub.-- CONTINUE.sub.-- STQSCAN); 295: } 296: else 297: { 298: V.sub.-- SPINLOCK(LOCK.sub.-- SLEEPTASK.sub.-- SPIN); 299: } 300: 301: return(LOCK.sub.-- INTERRUPTED); 302: } 303: 304: /* Check if we were woken up to perform deadlock search */ 305: if ((pss-->plocksleepq.sub.-- entry-->ststatus &

```
STINUSE) && 306: (pss-->plocksleepq.sub.-- entry-->ststatus & STCHECKDEADLOCK ))
307: { 308: /* 309: ** We need to initiate deadlock search. 310: ** First release
the bucket spinlock 311: */ 312: pss-->plockstat .vertline.= PL.sub.-- SLEEP 313:
V.sub.-- SPINLOCK(spinlock); 314: 315: P.sub.-- SPINLOCK(LOCK.sub.--
SLEEPTASK.sub.-- SPIN); 316: /* 317: ** Initiate deadlock search! 318: */ 319:
pss-->plocksleepq.sub.-- entry-->ststatus &= .about.STCHECKDEADLOCK ; 320: /* 321:
** Alarm handler might wakeup a task which has already 322: ** completed deadlock
search. So scan the queue for a task 323: ** which needs to perform deadlock search.
324: */ 325: if (pss-->plocksleepq.sub.-- entry-->ststatus & STDLSDONE) 326: { 327:
lock.sub.-- check timeout(LOCK.sub.-- CONTINUE.sub.-- STQSCAN); 328: 329: /* Go back
to sleep on semawait */ 330: goto sleepfor.sub.-- lock; 331: } 332: /* 333: **
Update the sequence number in pss so that the most 334: ** recent version of the
dependancy graph is checked for cycles. 335: ** Also increment the global sequence
number so that any sleep 336: ** task entry added after the search starts would have
a higher 337: ** sequence number than the initiator and hence would not be part 338:
** of the search. 339: */ 340: lockwait.sub.-- seqno = CIRCLE.sub.--
NEXT(Resource-->rdlc.sub.-- lockwait.sub.-- seqno); 341: /* 342: ** Reinitialize the
sequence numbers in the lock sleep task 343: ** queue if the current sequence number
cannot be compared 344: ** safely with the oldest sequence number in the lock 345:
** sleeptask queue. Since the lock sleeptask queue is ordered 346: ** on sequence
number, the oldest value in the queue would be 347: ** at the head of the queue.
348: */ 349: if (!CIRCLE.sub.-- SAFE( 350: ((SLEEPTASK *) (QUE.sub.-- NEXT
(&Resource-->rlocksleeptaskq))) 351: -->stsequence.sub.-- number, 352:
lockwait.sub.-- seqno)) 353: { 354: LOCK.sub.-- REINITIALIZE.sub.--
STSEQNO(lockwait.sub.-- seqno); 355: } 356: pss-->plocksleepq.sub.--
entry-->stsequence.sub.-- number = lockwait.sub.-- seqno; 357:
pss-->plocksleepq.sub.-- entry-->ststatus .vertline.= STDLSDONE;
```

Detailed Description Paragraph Table (14):

```
_____ typedef
struct sleeptask LINK stqueue; /* Link to the sleep task queue */ struct pss *stpss;
/* back link to the pss of the sleeping task */ struct semawait *stsemawait; /*
semawait task is sleeping on */ circ.sub.-- long stsequence.sub.-- number; /*
current value of lock wait sequence number */ long stunused1; /* Filler to match up
with the magic */ int16 ststatus; /* deadlock search status */ 0x5354 */stmagic; /*
"st" #if HAVE.sub.-- INT64 int32 stunused3; #endif /* HAVE.sub.-- INT64 */ long
stunused4; #if USE.sub.-- NATIVE.sub.-- TASKING syb.sub.-- event.sub.-- t event;
#endif /* USE.sub.-- NATIVE.sub.-- TASKING */ }SLEEPTASK; /* Status bits for the
sleep task entry. */ # define STDLSDONE 0x01 /* Deadlock search has been done for
this lock request once. */ # define STCHECKDEADLOCK 0x02 /* Task has to initiate
deadlock search. */ # define STINUSE 0x04 /* Pss-->pstentry is in use
_____ */
```

Detailed Description Paragraph Table (15):

```
_____ 1: /* 2:
** LOCK.sub.-- .sub.-- PERFORM.sub.-- DEADLOCK.sub.-- SEARCH 3: ** 4: ** int 5: **
lock.sub.-- perform.sub.-- deadlock.sub.-- search (SEMAWAIT *semawait, 6: ** LOCKREC
*lockrec) 7: ** Perform deadlock search by calling lock.sub.-- .sub.-- check.sub.--
deadlock. If 8: ** the deadlock search finds a victim cleanup the victim. If the 9:
** victim is us then release the lockrec also. 10: ** 11: ** Parameters: 12: **
semawait : the semawait which is causing the wait leading to 13: ** deadlock search
14: ** lockrec : lock request which is causing the wait leading to 15: ** deadlock
search 16: ** 17: ** Returns: 18: ** LOCK.sub.-- DEADLOCK - if the current task is
the deadlock victim 19: ** 0 - otherwise 20: ** 21: ** Side Effects: 22: ** Current
tasks lock sleep task entry might be moved to the 23: ** tail of the lock sleep task
queue. 24: ** Note that as part of deadlock victim cleanup, victims lock 25: **
sleep task entry could be deleted from the lock sleep task 26: ** queue. The lockrec
would be released if the current task 27: ** is the deadlock victim (all this would
be done by 28: ** lock.sub.-- cleanup.sub.-- deadlock.sub.-- victim called by this
routine) 29: ** 30: ** Called by: 31: ** lock.sub.-- .sub.-- semawait( ) 32: ** 33:
** MP synchronization: 34: ** LOCK.sub.-- SLEEPTASK.sub.-- SPIN is held when moving
the current tasks 35: ** lock sleep task entry to the tail. 36: ** 37: */ 38:
SYB.sub.-- STATIC int 39: lock.sub.-- perform.sub.-- deadlock.sub.-- search
(SEMAWAIT *semawait, 40: LOCKREC *lockrec) 41: { 42: LOCALPSS(pss); /* initialize
local copy of global pss */ 43: 44: /* place holder for call to deadlock search */
```

```
45: LOCKSLEEPOWNER waiter.sub.-- data; 46: /* deadlock table holding multiple
chains. */ 47: DLTAB deadlock.sub.-- tab; 48: 49: PSS *victim; /* deadlock victim */
50: 51: /* Record spid visited in */ 52: spid.sub.-- t dlcurrent.sub.--
path[SEMA.sub.-- MAX.sub.-- RECURSION] ; 53: /* deadlock search recursion*/ 54:
circ.sub.-- long lockwait.sub.-- seqno; 55: 56: #if !WORK.sub.-- GP.sub.-- PRIM 57:
MONEVENT.sub.-- LOCK.sub.-- DEADLOCK.sub.-- BEGIN( ); 58: MONITOR.sub.--
INC(mc.sub.-- lock(deadlock.sub.-- search)); 59: #endif /* WORK.sub.-- GP.sub.--
PRIM */ 60: 61: /* 62: ** Set up the waiting task data for the first call 63: ** to
deadlock search. 64: */ 65: waiter.sub.-- data.lsosemawait = semawait; 66:
waiter.sub.-- data.lsospinlock = semawait-->swspinlock; 67: waiter.sub.--
data.lsospid = pss-->pspid; 68: 69: /* 70: ** Clear the bitmap indicating visited
spids before starting 71: ** the deadlock search. 72: ** The number of bits in the
scanarray is 1 + MAXSPID because 73: ** spid's range from 1 to MAXSPID. We are
wasting zeroth bit. 74: */ 75: MEMZERO(Resource-->rdlc.sub.-- scanarray,
BYTESLEN(MAXSPID+1)); 76: 77: /* Initialize key info in the deadlock table. */ 78:
deadlock.sub.-- tab.dltxcess.sub.-- recur = FALSE; 79: deadlock.sub.--
tab.dltcycle.sub.-- count = 0; 80: 81: /* 82: ** Check for deadlocks involving
ourselves 83: ** Note that Resource-->rdlc.sub.-- sleepownerlist has enough space to
84: ** record two times configured number of pss. This would be 85: ** enough
because we break only two cycles so we would end up 86: ** recording the same task
at most twice. If we increase the 87: ** number of cycles detected by this algorithm
then we should 88: ** increase the size of Resource-->rdlc.sub.-- sleepownerlist.
89: */ 90: lock.sub.-- check.sub.-- deadlock(&waiter.sub.-- data, FIRST.sub.--
DEADLOCK.sub.-- CALL, 91: &deadlock.sub.-- tab, (LOCKSLEEPOWNER *) 92:
Resource-->rdlc.sub.-- sleepownerlist, dlcurrent.sub.-- path); 93: /* 94: ** If we
generate more than one deadlock, victimize the 95: ** initiator. 96: */ 97: if
(deadlock.sub.-- tab.dltcycle.sub.-- count > 1) 98: { 99: #if !WORK.sub.-- GP.sub.--
PRIM 100: MONITOR.sub.-- INC(mc.sub.-- lock(lock.sub.-- multipledeadlock)); 101:
#endif /* WORK.sub.-- GP.sub.-- PRIM */ 102: 103: if (pss-->pstat & P.sub.--
BACKOUT) 104: { 105: /* 106: ** If the initiator was a backout task and it was
involved 107: ** in more than one cycle then lock.sub.-- .sub.-- check.sub.--
deadlock would 108: ** have stored an alternate victim in the second deadlock 109:
** chain. Use that as the victim instead of the initiator. 110: */ 111: victim =
deadlock.sub.-- tab.dltchains[1] .dlcvictim.sub.-- pss; 112: } 113: else 114: { 115:
victim = pss; 116: } 117: } 118: else 119: { 120: victim = deadlock.sub.--
tab.dltchains[0] .dlcvictim.sub.-- pss; 121: } 122: 123: if (victim != pss) 124: {
125: /* 126: ** Move our sleep task entry from head to 127: ** tail because our
deadlock search is over and we 128: ** are not the victim. 129: */ 130: P.sub.--
SPINLOCK(LOCK.sub.-- SLEEPTASK.sub.-- SPIN); 131: lockwait.sub.-- seqno =
CIRCLE.sub.-- NEXT(Resource-->rdlc.sub.-- lockwait.sub.-- seqno); 132: LOCK.sub.--
.sub.-- MOVE.sub.-- TO.sub.-- TAIL(pss-->plocksleepq.sub. -- entry, lockwait.sub.--
seqno); 133: V.sub.-- SPINLOCK(LOCK.sub.-- SLEEPTASK.sub.-- SPIN); 134: } 135: 136:
/* Cleanup the victim if our search found a deadlock victim */ 137: if (victim) 138:
{ 139: #if !WORK.sub.-- GP.sub.-- PRIM 140: MONITOR.sub.-- INC(mc.sub.--
lock(deadlocks));
```

Detailed Description Paragraph Table (16):

```
141: #endif /* WORK.sub.-- GP.sub.-- PRIM */ 142: 143: /* 144: ** There is a
deadlock victim! 145: */ 146: Resource-->rdeadlock.sub.-- id++; 147: 148: /* Cleanup
victim */ 149: (void) lock.sub.-- cleanup.sub.-- deadlock.sub.-- victim(victim, 150:
&deadlock.sub.-- tab, lockrec); 151: } 152: 153: /* 154: ** Wake up the next person
in the sleep task queue to 155: ** continue the search if necessary. 156: */ 157:
P.sub.-- SPINLOCK(LOCK.sub.-- SLEEPTASK.sub.-- SPIN); 158: lock.sub.-- .sub.--
check.sub.-- timeout(LOCK.sub.-- CONTINUE.sub.-- STQSCAN); 159: 160: #if
!WORK.sub.-- GP.sub.-- PRIM 161: MONEVENT.sub.-- LOCK.sub.-- DEADLOCK.sub.-- END(0);
162: #endif /* WORK.sub.-- GP.sub.-- PRIM */ 163: /* 164: ** If we are the victim,
there is nothing more to do 165: ** except to return with DEADLOCK status. 166: */
167: if (victim == pss) 168: return LOCK.sub.-- DEADLOCK; 169: else 170: return 0;
171: }
```

Detailed Description Paragraph Table (17):

```
                                                                    1: /* 2:
** LOCK.sub.-- CHECK.sub.-- DEADLOCK 3: ** 4: ** void 5: ** lock.sub.-- check.sub.--
deadlock(LOCKSLEEPOWNER *waiter.sub.-- dlcheck.sub.-- data, 6: ** int level; short
*dlindex, DLTAB * deadlock.sub.-- tab, 7: ** LOCKSLEEPOWNER *sleepowner.sub.-- list,
```

8: ** spid.sub.-- t dlcurrent.sub.-- path[ ]) 9: ** 10: ** See if sleeping on a semawait would cause deadlocks. 11: ** The search algorithm detects deadlocks by detecting cycles in 12: ** a directed dependency graph amongst tasks waiting for locks. 13: ** In this graph there would be a directed edge from task A to 14: ** task B if task A was waiting for a lock that task B owns. The 15: ** dependency graph is not explicitly created. The algorithm 16: ** actually represents only the nodes in the graph in a list 17: ** called sleeping owners list. The edges are implicitly 18: ** represented through recursion. 19: ** 20: ** The algorithm accepts the spid of a waiting task and the lock 21: ** semawait it is waiting on as input (in waiter.sub.-- dlcheck.sub.-- data). 22: ** It first creates a list of owners of this lock which are 23: ** themselves sleeping for other locks (by calling 24: ** lock.sub.-- set.sub.-- sleeping.sub.-- owners.sub.-- list) . Then the algorithm is 25: ** repeated recursively on each of the sleeping owners until 26: ** either the list is empty or the search has exceeded 27: ** SEMA.sub.-- MAX.sub.-- RECURSION levels of recursion. 28: ** The recursion at any level stops when either the list is 29: ** empty or if one of the sleeping owners is the initiator 30: ** itself. In the latter case we have detected a deadlock and a 31: ** victim is selected to break the deadlock. 32: ** 33: ** The algorithm only detects deadlocks involving the task which 34: ** initiated the search. It detects upto two cycles involving 35: ** the initiator. The search is stopped after detecting two 36: ** cycles to contain the cost of search. 37: ** 38: ** This routine does not acquire any spinlocks since: 39: ** 40: ** 1. The spinlocks are needed only when we traverse the 41: ** lock hash chains or the lock sleep task queue to 42: ** record sleeping owners. A11 this is done in 43: ** lock.sub.-- set.sub.-- sleepingowner.sub.-- list. 44: ** 2. The algorithm allows the lock manager data structures 45: ** to change while deadlock search is in progress. 46: ** 3. Only one task is allowed to perform deadlock search at 47: ** any time; 48: ** One interesting fallout of this observation is that this 49: ** task can yield now if it has almost exhausted its time 50: ** quantum. 51: ** We can detect a tree of deadlock chains; for instance 1.2.3 52: ** and 1.2.4 are two deadlock chains which have the first two 53: ** dlcitem's in common. We record the entire first deadlock 54: ** information (1.2.3) in the 1st DLCHAIN and the entire second 55: ** deadlock in the 2nd DLCHAIN. This algorithm could support 56: ** more than 2 DLCHAINs, if later we decide to kill a list of 57: ** victims instead of the caller. 58: ** 59: ** We will resolve only deadlock chains starting at level 0. The 60: ** algorithm would detect deadlocks starting at other levels but 61: ** would not try to break the deadlocks. This is done to limit 62: ** 63: ** the cost of deadlock search. Note that the dlcinfo structure 64: ** in each dltchain is filled up as soon as the deadlock is 65: ** detected (lock.sub.-- fill.sub.-- dltab( ) is called once to fill in the 66: ** information. 67: ** 68: ** Parameters: 69: ** waiter.sub.-- dlcheck.sub.-- data : Information recorded for the waiter by 70: ** the caller. 71: ** level : Level of recursion (used to prevent excessive 72: ** recursion 73: ** dlindex : Index into deadlock chain structures off of 74: ** deadlock.sub.-- tab. 75: ** deadlock.sub.-- tab : table holding deadlock information about 76: ** possible 77: ** multiple deadlock chains 78: ** sleepowner.sub.-- list : Pointer to record the lock sleep owner 79: ** list at this level. 80: ** dlcurrent.sub.-- path : Array of spids involved in this cycle in 81: ** the order they were visited. 82: ** 83: ** Returns: 84: ** Nothing. 85: ** 86: ** Side Effects: 87: ** dlindex contains the number of deadlocks found, if it 88: ** contains 0, there is no deadlock dectected. 89: ** deadlock.sub.-- tab contains the list of victim pss. 90: ** 91: ** Fields affected in deadlock.sub.-- tab: 92: ** dltchains (mapped to deadlock.sub.-- chain) 93: ** are used to track deadlock info for each 94: ** deadlock.

Detailed Description Paragraph Table (18):
95: ** dltxcess.sub.-- recur may get set if we exceed the max 96: ** recursion level 97: ** 98: ** Fields affected in each deadlock.sub.-- chain 99: ** dlcvictim.sub.-- pss is used to communicate the detection of a 100: ** deadlock in lock.sub.-- check.sub.-- deadlock() 101: ** If we are recursing in 102: ** lock.sub.-- check.sub.-- deadlock( ), this value may 103: ** not be the final victim. 104: ** dlclast.sub.-- item is incremented for each collection of 105: ** deadlock info 106: ** in the different levels of recursion in 107: ** lock.sub.-- check.sub.-- deadlock( ). 108: ** 109: ** Called by: 110: ** lock semawait( ), lock.sub.-- check.sub.-- deadlock ( ) 111: ** 112: ** MP synchronization: 113: ** None! 114: ** 115: */ 116: SYB.sub.-- STATIC void 117: lock.sub.-- check.sub.-- deadlock(LOCKSLEEPOWNER *waiter.sub.-- dlcheck.sub.-- data, 118: int level, 119:

```
DLTAB * deadlock.sub.-- tab, 120: LOCKSLEEPOWNER *sleepowner.sub.-- list, 121:
spid.sub.-- t dlcurrent.sub.-- path[ ]) 122: { 123: LOCALPSS (pss) ; /* initialize
local copy of global Pss */ 124: REGISTER PSS *ownerpss; 125: PSS *victimpss; /* Pss
ptr of victim found in search */ 126: DLCHAIN *deadlock.sub.-- chain;/* local ptr to
a deadlock.sub.-- chain. */ 127: int32 ownerpid; 128: LOCKSLEEPOWNER
*newsleepowner.sub.-- list; 129: int noofsleepers; 130: int i; 131: REGISTER PSS
*tmp.sub.-- pss; 132: CFG.sub.-- VALUE cfgprtdlockinfo; 133: 134: /* 135: ** Check
for stack overflow in this recursive routine. 136: ** This must appear directly
after the declaration section 137: ** in the routine. 138: */ 139: CHECKSTACKOFLOW;
140: 141: /* 142: ** At each level check the timeslice and yield if need to! 143: **
The state information is available in lock sleep owner list and 144: ** we allow the
lock manager state to change after recording 145: ** the waiter data. 146: */ 147:
if ( pss-->ptimeslice < 0) 148: { 149: (void) upyield( ); 150: } 151: 152: (void)
cfg.sub.-- getconfig(CFG.sub.-- PRTDEADLOCKINFO, CFG.sub.-- RUNVALUE, 153:
&cfgprtdlockinfo) 154: 155: /* set the deadlock chain pointer, and the index on
dlcitem's within 156: ** that chain 157: */ 158: deadlock.sub.-- chain = &
(deadlock.sub.-- tab-->dltchains [deadlock.sub.-- tab-->dltcycle.sub.-- count]);
159: deadlock.sub.-- chain-->dlcvictim.sub.-- pss = 0; 160: 161: /* 162: **
Terminate recursion if excessive. Nominate current process as the 163: ** victim if
this level is reached. 164: */ 165: if (level == SEMA.sub.-- MAX.sub.-- RECURSION)
166: { 167: /* 168: ** Set deadlock.sub.-- tab-->dltxcess.sub.-- recur to true so we
can 169: ** terminate the search for a victim. 170: ** Lock.sub.-- print.sub.--
deadlockchain( ) uses this info to identify 171: ** phony deadlocks. Don't copy any
info for this level. 172: */ 173: deadlock.sub.-- tab-->dltxcess.sub.-- recur =
TRUE; 174: deadlock.sub.-- tab-->dltcycle.sub.-- count++; 175: 176: /* victimize pss
of requestor. */ 177: deadlock.sub.-- chain-->dlcvictim.sub.-- pss = pss; 178: /*
179: ** If initiator is a backout task then select the 180: ** first non BACKOUT
task in the path. 181: ** If there is no non BACKOUT task then Tough Luck!! 182: **
In this case we have to choose the initiator as the 183: ** victim. 184: */ 185: if
(pss-->pstat & P.sub.-- BACKOUT) 186: { 187: for(i=0; i<= level; i++) 188: { 189:
tmp.sub.-- pss = GET.sub.-- UNKEPT.sub.-- PSS(dlcurrent.sub.-- path[i]); 190: if (!
(tmp.sub.-- pss-->pstat & P.sub.-- BACKOUT)) 191: { 192: deadlock.sub.--
chain-->dlcvictim.sub.-- pss = tmp.sub.-- pss; 193: break; 194: } 195: } 196: } 197:
return; 198: } 199: 200: /* Store the waiters spid in the current path array. */
201: dlcurrent.sub.-- path[level] = waiter.sub.-- dlcheck.sub.-- data-->lso spid;
202: /* 203: ** Create the sleeping owners list. 204: ** lock.sub.-- set.sub.--
sleepingowner.sub.-- list returns the number of sleeping 205: ** owners that it
recorded. Advance the sleeping owner list 206: ** pointer by these many entries to
pass to the next level. 207: */ 208: noofsleepers 209: = lock.sub.-- set.sub.--
sleepingowner list(waiter.sub.-- dlcheck.sub .-- data, 210: sleepowner.sub.-- list);
211: newsleepowner.sub.-- list = sleepowner.sub.-- list + noofsleepers; 212: 213: /*
214: ** Recurse for each sleeping owner. 215: */ 216: for *(; (ownerpid =
sleepowner.sub.-- list-->lsospid) 217: != ENDOFLSOLIST ; sleepowner.sub.-- list++)
218: { 219: /* Grab the sleeping owner pss */ 220: ownerpss = GET.sub.--
UNKEPT.sub.-- PSS(ownerpid); 221: /* 222: ** As a last check before recursing see if
this task is no 223: ** longer sleeping. If it is not then we do not need to 224: **
recurse on this entry. This is done without holding the 225: ** bucket spinlock.
226: */ 227: if (ownerpss-->pwaitsema == o) 228: { 229: continue; 230: } 231: 232:
/* We have a cycle if we encounter our own pss again */ 233: if (ownerpss == pss)
234: { 235: 236: # ifdef TRACE.sub.-- LOCK 237: if ( (TRACECMDLINE(LOCKM, 4))
.vertline. .vertline. 238: cfgprtdlockinfo.int32.sub.-- value) 239: { 240: /* Fill
all the information about this chain. */ 241: lock.sub.-- fill.sub.--
dltab(deadlock.sub.-- chain, dlcurrent.sub.-- path, 242: level, ownerpid); 243: }
244: # endif /* TRACE.sub.-- LOCK */ 245: 246: /* 247: ** We found a new deadlock
increment the cycle.sub.-- count.
```

Detailed Description Paragraph Table (19):

```
248: */ 249: deadlock.sub.-- tab-->dltcycle.sub.-- count++; 250: 251: /* stop
looping if we've reached the limits of detection */ 252: if (deadlock.sub.--
tab-->dltcycle.sub.-- count >= MAX.sub.-- VICTIMS) 253: { 254: /* Choose the
initiator as the victim */ 255: deadlock chain-->dlcvictim.sub.-- pss = pss; 256: /*
257: ** If initiator is a backout task then select the 258: ** first non BACKOUT
task in the path. There has to be a 259: ** non BACKOUT task in the cycle because:
260: ** 1. known access method assumptions imply that we can 261: ** not have a
cycle involving just backout tasks. 262: ** 2. multiple cycle requires a shared lock
```

and BACKOUT 263: ** tasks acquire only address locks. So there has to 264: ** be a non BACKOUT task in this case. 265: */ 266: if (pss-->pstat & P.sub.-- BACKOUT) 267: { 268: for(i= 0; i<= level; i++) 269: { 270: tmp.sub.-- pss = GET.sub.-- UNKEPT.sub.-- PSS(dlcurrent.sub.-- path[i]); 271: if (! (tmp.sub.-- pss-->pstat & P.sub.-- BACKOUT)) 272: { 273: deadlock.sub.-- chain-->dlcvictim.sub.-- pss = tmp.sub.-- pss; 274: break; 275: } 276: } 277: } 278: return; 279: } 280: /* 281: ** We found a new deadlock. Select a deadlock victim for 282: ** this chain. 283: */ 284: lock.sub.-- select.sub.-- deadlock.sub.-- victim(deadlock.sub.-- chain, 285: dlcurrent.sub.-- path, 286: level) 287: 288: /* 289: ** Clear the visited bit for all the tasks involved in this 290: ** cycle so that we can catch other cycles involving the 291: ** same tasks. 292: */ 293: for (i=0; i<= level; i++) 294: { 295: CLEAR.sub.-- BIT (Resource-->rdlc.sub.-- scanarray,dlcurrent.sub.-- path[i]); 296: } 297: /* 298: ** Clear the visited bit of the remaining tasks in the 299: ** sleeping owner list for this level. Leaving their bits 300: ** on would prevent them from being recorded at other 301: ** levels. Since we have not visited these tasks yet, they 302: ** might be part of deadlocks involving the initiator. 303: ** Clearing the bits ensures that they would be recorded 304: ** and visited if encountered at other levels. 305: */ 306: for (sleepowner.sub.-- list++; 307: (ownerpid = sleepowner.sub.-- list-->1sospid) !=ENDOFLSOLIST; 308: sleepowner.sub.-- list++) 309: { 310: CLEAR.sub.-- BIT(Resource-->rdlc.sub.-- scanarray,ownerpid); 311: } 312: /* 313: ** We can move return to previous level of recursion since 314: ** all deadlocks involving tasks below this level on this 315: ** path can be broken by selecting any one of the tasks in 316: ** dlcurrent.sub.-- path as the victim. This is because all such 317: ** deadlocks would have to involve all the tasks in 318: ** dlcurrent.sub.-- path also. Since we have already chosen one of 319:

CLAIMS:

1. In a multi-tasking database system having a server storing a database connected to a plurality of clients, said database system providing a plurality of database engines for processing requests from the clients for database operations, an improved method for providing access to objects provided by the database system, the method comprising:

providing each database engine with a lock manager for controlling access to objects in the database system;

storing lock management data structures which can be shared among all database engines, said lock management data structures storing locking information about a first plurality of locks that comprise a plurality of lock types for protecting access to objects in the database system which are shared;

controlling access to said lock management data structures through at least one hash table comprising a plurality of hash buckets, each hash bucket being associated with a particular lock type from said plurality of lock types provided by the system;

protecting access to said lock management data structures themselves with a second plurality of locks that comprise spin locks; and

providing parallel access to different ones of said first plurality of locks provided by the database system by associating each spin lock from said second plurality of locks with a particular group of hash buckets.

2. The method of claim 1, wherein each database engine operates on a separate processor of a symmetric multi-processor computer system.

5. The method of claim 3, wherein each lock granted from said first plurality of locks can be either an exclusive lock or a shared lock.

9. The method of claim 8, wherein locks from said first plurality of locks which can coexist include shared locks.

12. The method of claim 1, further comprising:

receiving a request for access to a particular object;

in response to said request, examining the hash bucket associated with the particular object for determining whether a lock from said first plurality of locks exists to protect the object;

if a lock does not already exist to protect the object, creating a lock object storing locking information about the particular object and linking that lock object onto the hash bucket; and

posting a lock request for access to the particular object by appending the lock request to the lock object if no other lock requests exist for the particular object, or appending the lock request to the end of the last lock request which has been previously appended.

21. In a multi-tasking database system, in which tasks may deadlock on one another while waiting for locks for accessing objects to be granted, said locks being created from a plurality of transactional lock types available in the system for protecting objects having shared access, an improved method of deadlock detection, the method comprising:

upon receiving a request from a certain task for a particular lock which can not yet be granted from said locks, placing the certain task to sleep while waiting for the particular lock to be granted;

setting an alarm for periodically determining which task has been sleeping the longest on a lock;

if the particular lock can be granted from said locks, awakening said certain task for continuing execution with access to the object controlled by the particular lock; and

if the alarm triggers before the certain task has been granted the particular lock, awakening the certain task for undertaking deadlock detection.

29. The method of claim 28, further comprising:

detecting a deadlock upon finding a cycle in said directed dependency graph, said cycle representing a dependency of a particular task which cycles through other tasks back to the particular task.

(2)

WEST

End of Result Set

☐ Generate Collection Print

L39: Entry 1 of 1 File: USPT Jun 29, 1999

DOCUMENT-IDENTIFIER: US 5918248 A
TITLE: Shared memory control algorithm for mutual exclusion and rollback

Abstract Text (1):
The invention provides a mechanism for allowing a share memory/parallel processing architecture to be used in place of a conventional uni-processing architecture without requiring code originally written for the conventional system to be rewritten. Exclusive Access and Shared Read Access implementations are provided. A rollback mechanism is provided which allows all the effects of a task to be undone.

Brief Summary Text (2):
The invention relates to shared memory systems for use in parallel processing environments.

Brief Summary Text (5):
One of the major problems in a multiprocessor system is in preventing data access collisions due to two or more processors accessing the same data at the same time. A data collision occurs when multiple processors interleave accesses to the same data structure such that an inconsistent state is read or updates are lost. For example, if one program makes multiple reads from a data structure while another program, executing concurrently with the first program, modifies the structure such that some reads are made before the structure is modified, and some after, this would result in an inconsistent state of the data structure being read. Typically in multiprocessor architectures, the software is specifically designed from the start with explicit knowledge of this condition in the system and is therefore designed in such a way as to avoid the problem. Mechanisms for doing this generally provide exclusive access to the memory subject to such collisions via software semaphore techniques, or bus lock techniques. These techniques prevent interleaving accesses to data, and require explicit software knowledge of the nature of collisions and specific mechanisms for avoiding them.

Brief Summary Text (6):
Correct operation of many conventional shared memory multiprocessor architectures requires measures to ensure cache coherency. If one or more processors have a copy of an item from shared memory and one of the processors modifies that item, then the modification must be propagated to the other processors. Cache coherency implementations typically require complex high speed protocols between processors and caches.

Brief Summary Text (10):
It is an object of the invention to provide a parallel processor/shared memory architecture which obviates or mitigates one or more of the above identified disadvantages.

Brief Summary Text (11):
According to a first broad aspect, the invention provides a parallel processing/shared memory system comprising: a plurality of processors for running a plurality of tasks each identifiable by a task identifier; one or more memory modules each having a plurality of memory locations, each memory location associatable with one of the task identifiers; means for allowing or denying a particular task to access a particular memory location on the basis of the task

identifier associated with that location and task identifier of the particular task, and for associating the task identifier of the particular task with the memory location when the particular task is allowed access to that location.

Brief Summary Text (14):
Preferably, memory locations include standard locations and shared read locations, and any task is allowed read access to a shared read location, but only a single task is allowed write access to a shared read location.

Brief Summary Text (15):
An advantage provided by the data ownership aspect of the invention is that a single processor architecture may be replaced with a multi-processor parallel processing architecture without requiring the application software to be rewritten to function properly.

Brief Summary Text (16):
An advantage of the multiprocessor/shared memory architecture according to this invention is that the cache coherency problem referred to above does not arise, because data ownership ensures that only one task and consequently, only one processor can own a data location.

Drawing Description Text (3):
FIG. 1 is a block diagram of a shared memory/multiprocessor architecture;

Drawing Description Text (4):
FIG. 2 is a block diagram of the memory ownership control system according to the invention for an Exclusive Access implementation;

Drawing Description Text (9):
FIG. 7 is a flowchart of the steps executed by the shared memory/multi-processor architecture during a memory access;

Drawing Description Text (12):
FIGS. 10a-10d are timing diagrams for the timing of update tasks for Shared Read implementations.

Drawing Description Text (13):
FIG. 11 is a block diagram of the memory ownership control system according to the invention for an implementation allowing both Exclusive Access and Shared Read Access;

Drawing Description Text (14):
FIG. 12 is the record structure for the tag memory of FIG. 11 which includes fields for the Shared Read Access functionality; and

Drawing Description Text (15):
FIG. 13 is the record structure for the SST memory of FIG. 11 which includes fields for the Shared Read Access functionality.

Detailed Description Text (2):
Conventional uni-processor architectures allow only one process to be run at a time, although the logical behaviour of parallel operation can be achieved by running each of a number of processes in turn for a short period or burst of operation. An operating system controls the swapping in and out of the various processes. A time slice is a unit of time which generally corresponds to the permitted duration of a burst. A time slice may be lengthened by delaying the end of the time slice beyond the normal permitted duration, in appropriate cases.

Detailed Description Text (3):
The period of time (referred to above as a burst) that a process runs continuously is referred to as a task. A process is made up of an arbitrary number of tasks. Because one burst runs at a time, processing during a burst is atomic on conventional uni-processor architectures. During the burst, the task may make changes to shared memory or registers, but no other task can see the effects until the burst is completed. Although a task is atomic, this does not mean that a process

is atomic; a long running process is split into tasks, and other processes run in the intervals between the tasks. These intervals occur at arbitrary points in the process that are controlled by the process, and more specifically by the tasks themselves, so it can be assumed that any given task forming part of a process will fall into a single burst and operate atomically.

Detailed Description Text (4):
Many processes may be event driven. An event is a stimulus from outside the process, such as a message from a peripheral or from another process. When an event occurs, the process is started, runs briefly, and then stops to wait for the next event. If it can be ensured that a given process run will complete within a single burst, then the event processing can be considered atomic.

Detailed Description Text (7):
Two broad implementations of the invention will be described. One implementation is referred to as an "Exclusive Access" implementation, and the other implementation is referred to as a "Shared Read Access" implementation. In an Exclusive Access implementation, a particular task takes ownership of a memory location upon either a read or a write access, while in a Shared Read Access implementation, a particular task owns a memory location only upon a write access.

Detailed Description Text (8):
The first implementation to be described will be the Exclusive Access implementation. Referring now to FIG. 1, a parallel processor/shared memory architecture has a plurality of processors or processing elements 10 (only three of which are shown) connected through an interconnect 18 to a main shared memory consisting of one or more memory modules 20 (only one shown). One or more I/O devices 24 are also connected to the interconnect 18 (only one shown). It is common to equip systems with redundant hardware components to aid in fault tolerance, but these are not essential to the invention and are not shown. Each processor 10 has a respective cache memory 11 having a cache line size of 32bytes, for example. Each memory module 20 has a data memory and has memory ownership control functionality implemented with hardware. For ownership purposes, the data memory in each memory module 20 is divided up into segments of memory referred to as "lines" or "memory locations" each of which is preferably 32 bytes in size, the same size as a cache line. In addition, for each 32 byte data location, the data memory has another 32 bytes of memory for storing a second copy of the data. The two copies of the data will be referred to as "Copy A" and "Copy B". For brevity, a multi-processor/parallel processing architecture such as that illustrated in FIG. 1 will be referred to as an "MPA" (multi-processor architecture) while a uni-processor architecture will be referred to as a "UPA" (uni-processor architecture).

Detailed Description Text (9):
The shared memory 20 of an MPA according to the invention logically corresponds to the memory of a conventional UPA. Each processor 10 in an MPA can run a separate task. The task operates on one of the processors 10, taking data as required from the shared memory 20. The processors 10 will maintain copies of the data in their respective cache memories 11, eventually flushing all their results to the shared memory 20. This allows several tasks to proceed in parallel. An OS (operating system) runs on the processors 10 and controls on which processor a given task is to be run. An OS designed for a UPA may be changed to manage tasks in this parallel manner, with little or no change to the application code originally written for a UPA. Each task running on one of the processors 10 is identified by a unique SID (slot identification number). There must be sufficient unique SIDs to ensure that a processor can be allocated one when required. If each processor can run N tasks simultaneously (this is also referred to as each processor having N "slots") and there are M processors, then there must be at least N.times.M unique SIDs. For this description of the preferred embodiment, it is assumed that there are 16 processors, each of which may require up to 16 SIDs, meaning that there is a requirement for 256 unique SIDs.

Detailed Description Text (10):
Special arrangements must be made to ensure that tasks on the MPA will operate as if they are atomic. More precisely, results from an MPA task must not be affected by the presence of other tasks running at the same time. FIG. 2 is a block diagram of

the memory ownership control hardware which forms part of the shared memory 20. The memory ownership control hardware together with the operating system running on the processors controls whether a given task running on one of the processors will be allowed to continue after attempting to access a particular memory location in the shared main memory. It is noted that the majority of functions required to control memory ownership are implemented in the memory ownership control hardware forming part of the memory modules without the involvement of the operating system. The operating system may get involved in cases in which contention for access to a particular memory location occurs, as described in detail below.

Detailed Description Text (11):
The memory ownership control hardware includes three memories, namely a SST (slot state) memory 26, a tag memory 27, and a TST (TIN state) memory 28. In addition to the three memories 26,27,28, the hardware also includes ownership control logic which performs the actual control over access to and ownership of the data locations. Some of the control logic is shown explicitly in FIG. 2, and some of it is lumped together into blocks 29 and 30 entitled the OCL (ownership control logic) and the scrub hardware respectively.

Detailed Description Text (12):
The operating system controlling the processors 10 of FIG. 1 is shown in various locations in FIG. 2 in dotted circles labelled OS, but the OS does not form part of the memory ownership control hardware. The memory ownership control hardware forming part of the memory module communicates with the OS through the interconnect 18 (FIG. 1). It is to be understood that FIG. 2 is a functional block diagram, and that an efficient implementation would combine all of the logic shown into a minimal number of chips, preferably only one. Thus, while many connecting lines are shown, in many cases these simply represent information flow between functional blocks, and not actual physical connections between discrete components. Furthermore, not all information flows are represented explicitly by lines in FIG. 2.

Detailed Description Text (13):
The SST memory 26 contains 256 records, one for each possible SID. Each time a task having a SID is started, it is allocated a TIN (task identification number) in each memory module. When the task is finished, the SID is immediately available for identifying another task. However, the TIN is not made available to another task until all the effects of running the task have propagated throughout the memory modules and the memory control system, as discussed in detail below. The tag memory 27 contains a tag record for each line of memory in the memory module, and contains ownership information for that line. The ownership information includes the TIN of the task which owns that line of memory. The TST memory 28 contains state information for each TIN. Each of these memories will be discussed in detail below.

Detailed Description Text (14):
Inputs to the SST memory 26 include a next available TIN 31, OS primitives 32, and a SID input 34 for inputting the SID of the accessing task. Outputs from the SST memory 26 include a SST TIN output 35, and a TIN state change command 37. It is noted that various different OS primitives may be generated by the OS, and that various blocks in the memory ownership control hardware will be effected by these OS primitives. Collectively,.they are shown in the block diagram as OS primitives 32.

Detailed Description Text (15):
The structure of each record in the SST memory 26 is shown in FIG. 3. Each record includes an "Active" field, and a "TIN" field. The active field is "0" if the slot does not have a task running in it, and contains a "1" if the slot does have a task running in it. The TIN field contains the TIN currently assigned to this slot. When the OS starts a task on a given processor, the OS allocates the next available SID for that processor. It also instructs the SST memory 26 in each memory module 20 through a "task launch" OS primitive 32 to allocate the next available TIN 31 for that SID, and the Active field in the SST memory record for that SID is set. The next available TIN 31 is received from the TST memory 28 and recorded in the TIN field in the SST memory record for that SID. The "task launch" is a broadcast type command in the sense that it is not memory module specific.

Detailed Description Text (19):

The procedure for controlling access to a memory location will now be described with reference to the flowchart in FIG. 7, and to the previously described FIGS. 1-6. Block numbers refer to the flowchart in FIG. 7. When a task having a particular SID attempts to access a particular location, the relevant processor passes the memory address as input 38 to the tag memory 27. The tag memory 27 outputs the TIN (if any) stored in the tag for that memory address as TIN output 40. The OS also passes the SID 34 of the accessing task to the SST memory 26 which looks up the corresponding TIN if any, and produces this as the SST TIN output 35. A check is made of the Owned field in the corresponding record in the tag memory (block 100). If the Owned field indicates that the location is unowned, then the task is allowed access to the location, and the Owned field is set to indicate that it is owned (block 102). If the owned field indicates that the location is owned, then a comparison between the TIN of the accessing task and the task which owns the location is made (block 104). This amounts to the comparator 49 comparing the TINs on the two TIN outputs 35,40. The OCL 29 receives the comparator 49 output. If these are the same (yes path, block 104), then the task is allowed access. If it is a read access (yes path,block 106) then the location is read (block 108) and the contents passed to the accessing task. If it is a write access (no path, block 106) a check is made to determine if the location was previously modified (block 110) by checking the Dirty field in the tag. If it was not previously modified (no path, block 110), then the Active Copy field is flipped (block 112). In other words, if the Active Copy field previously indicated Copy A (Copy B) as the active copy, then it is flipped to indicate Copy B (Copy A) as the active copy, thereby preserving the contents of Copy A (Copy B). This amounts to toggling the bit in the Active Copy field. After this, the Dirty field will be set to indicate that the location has been modified (block 114). After this, (or if the location had been previously modified, yes path--block 110) the task is allowed write access to the location, and the new data is written to the active copy (block 116). The tag memory 27 is then updated by writing the TIN of the accessing task to the Owner field if necessary (block 118).

Detailed Description Text (20):
If the accessing TIN 35 and the owner TIN 40 are not the same, (no path, block 104), then a check of the state of the owner TIN is made in the TST memory 27 (block 120). The owner TIN 40 is passed to the TST memory 28 through multiplexer 52, and the TST memory looks up the state for that TIN. If the state of the owner TIN is "Active" (yes path, block 120) then the task is denied access, and a "Blocked" state signal 53 is returned (block 122). If the state of the owner TIN is not "Active" (no path, block 120) then a check is made to see if the state is "Committed" (block 124). If it is committed (yes path, block 124), then the tag field is initialized (block 126) by clearing the Dirty flag. The task is then allowed access as before, continuing from block 102.

Detailed Description Text (21):
If the state is not "committed" (no path, block 124) then the state must be "rollback". A check is made to determine whether the particular location was modified by the task which previously owned it (block 128). This is done by checking the Dirty flag in the relevant tag record. If the location was not modified (no path, block 128), then the task is allowed access as before, continuing from block 126. If the location was modified (yes path, block 128), then the active copy field is flipped to point to the copy of the memory location which was not modified by the task which previously owned it (block 130). This amounts to toggling the Active copy field. After this, access is permitted as before, continuing from block 126.

Detailed Description Text (23):
The state transitions which occur for each TIN can be modelled with a state machine, as depicted in FIG. 8. Similarly, the state transitions which occur for each memory location can be modelled with a state machine as depicted in FIG. 9. Initially, each TIN is INACTIVE (bubble 200) since there is no task associated with it, and each memory location is in an UNOWNED state (bubble 210), or is owned by another task. When a new task starts, the "task launch" primitive 32 instructs the SST memory 26 to assign the next available TIN to the SID of the new task. The TIN is written into the SST memory 26 for that SID. At the same time, the SID is written into the TST memory 28 for the newly assigned TIN. The Active field in the SID record then changes from INACTIVE to ACTIVE, and the state bits in the TIN record in the TST memory are changed to indicate Running. At this point the TIN state transition

INACTIVE (bubble 200) .fwdarw.ACTIVE (bubble 202) has occurred. No other state changes from the INACTIVE state are possible as can be seen in FIG. 8.

Detailed Description Text (31):
As mentioned previously, the "BUSY" bit (the bold bits in FIG. 5) is used to control TIN reallocation. TINs are allocated by each memory module sequentially from a large table. When a task having a particular TIN is completed, or is rolled back, the state in the TST memory 28 is updated to reflect this, but the individual tags in the tag memory 27 will not be updated until they are either visited by the scrub process, or are accessed by another task.

Detailed Description Text (33):
A task starts, and is allocated a TIN at time T. At time T+t.sub.A, the task is checked by the scrub hardware 30 to see if it is still running: t.sub.A is chosen so that most tasks commit or rollback before reaching T+t.sub.A. Assume that it takes a time t.sub.B, to complete a scrub cycle. In other words, every tag will be updated within a time interval of length t.sub.B. Therefore at time T+t.sub.A +t.sub.B, the scrub process will have completed a cycle, so if the TIN was not running at time T+t.sub.A it is ready to be reallocated at time T+t.sub.A +t.sub.B. It also takes a time t.sub.C to cycle through the TIN table and return to the same TIN. The time t.sub.C is a function of the frequency with which new tasks are started as detailed below, and of the size of the TIN table. If the task was not running at time T+t.sub.A, it can certainly be reallocated on the next cycle at time T+t.sub.C since t.sub.C >t.sub.B +t.sub.A.

Detailed Description Text (34):
On the other hand, if the task was still running at time T+t.sub.A, it might not be completely scrubbed in time for the next cycle. In this case, the TIN is marked BUSY by setting the corresponding BUSY bit in the TST memory 28 at time T+t.sub.A. At time T+t.sub.C, the task is due to be reallocated, but because it is marked BUSY this reallocation is skipped and the next available TIN is allocated. The old task may or may not still be running at this time, but is treated just like a newly started task. After a further time t.sub.A the task is checked again, and has another chance to be found not running and to become available for reallocation. In the general case, a task starts at time T, is checked at times T+kt.sub.C +t.sub.A until it is found not running, and is eventually reallocated at the next T+kt.sub.C +t.sub.A after that.

Detailed Description Text (35):
It is preferred that rather than measuring or timing directly all the time intervals t.sub.A,t.sub.B,t.sub.C, time is measured in terms of the numbers of tasks started. Assume initially that new tasks are started at regular intervals V and that the TIN table has N entries. All task allocation arithmetic is performed modulo N, and wraps around at the end of the TIN Table.

Detailed Description Text (36):
Consider task N which started at some known time. Some time later, task number N+X is started. By this time, task N has been running for a period of at least X.times.V. For properly chosen X, therefore, task number N should have either committed or rolled back before task number N+X is allocated. The state of task number N is examined when task number N+X is allocated; if N is still running, its BUSY bit is set, otherwise its BUSY bit is reset.

Detailed Description Text (37):
Still later, task number N+X+Y is allocated. By this time, a further period Y.times.V has elapsed. For properly chosen Y, this interval is sufficient for the scrub process to complete one pass through the memory, so during the interval every possible reference to task N has been visited by the scrub process. If N was not running at the start of the interval, all references to N have now been resolved. In other words, if the BUSY bit task N is not set at the beginning of the period Y.times.V then task N becomes available for reallocation at the end of that period.

Detailed Description Text (38):
It only remains to ensure that the size of the TIN table is greater than X+Y, so that a task N becomes available as described before the allocation wraps around.

Detailed Description Text (40):
For practical use, the mechanism must allow tasks to start at somewhat irregular intervals. A leaky bucket mechanism is used. This enforces a maximum sustainable task allocation rate of one task every time V, but allows up to K tasks to be started in advance of the times so defined. Conversely, task starts may be delayed, but the opportunity so lost cannot be regained.

Detailed Description Text (41):
More exactly consider a buffer which can hold up to K tokens. A token is added to the buffer every time V unless the buffer is already full. A token is removed from the buffer whenever a task is started or a busy task is skipped. If the buffer is empty task allocation must wait until a new token becomes available.

Detailed Description Text (42):
The TIN table must provide space for an extra K tasks. This merely requires that the size of the TIN table should be greater than X+Y+K.

Detailed Description Text (43):
Data ownership provides a running task with exclusive access to its data for the duration of the task, so that the task will operate on all data atomically. It is noted that, in general, it does not permanently associate data with a process or a subsystem or ensure that object data structures are accessible only through the correct methods. Long term data protection and making tasks atomic are quite separate concepts, although there are some apparent similarities.

Detailed Description Text (47):
Typically, the OS controlling the MPA maintains a list of tasks which are ready to run. When processing resources become available, the OS selects a task from the list, and starts the task running on a processor. A task normally runs until it completes its work, or until it exhausts the time allocation permitted by the OS. At this point, control returns to the OS. If the task completes its work it returns control to the OS; if the time allocation is exhausted then a timer interrupt forces a return to OS.

Detailed Description Text (53):
When control returns to the OS, the OS kernel calls on data ownership mechanisms to commit or rollback the task, updates its lists of tasks to be run, and schedules a new task. In some circumstances the OS may retain a partially completed task in a suspended state and continue it later. A suspended task is also said to be "blocked".

Detailed Description Text (54):
Normally, the OS runs one task at a time on each processor, and the task either runs to completion and commits, or it suffers a failure or collision and is rolled back. As indicated previously, the OS may suspend or block a task in which case it saves the task context, and runs another task on the processor.

Detailed Description Text (55):
When a task on a processor is forced to block, the registers and data used by the task are saved and the cache is flushed. The processor may then be used for other tasks. Eventually, the saved context is restored, and the blocked task can continue; otherwise the task can be rolled back. In a preferred implementation, that the blocked task continues on the same processor and slot; it cannot migrate to a different processor.

Detailed Description Text (56):
It is possible for a set of blocked tasks to deadlock, so that every task in the set must wait for another task in the set to unblock first. The OS keeps track of blocked tasks, and resolves deadlocks by rolling back one or more of the tasks involved. It is not desirable to have a large number of blocked tasks in the system. Tasks normally run for a short time before releasing ownership. If a task is blocked, it will retain ownership of any data which it has already accessed, so if too many tasks are blocked then there will be a large amount of inaccessible data, possibly increasing the number of collisions. Because of this, it is preferred that

the OS block tasks in some exceptional circumstances but to more commonly rely on rollback.

Detailed Description Text (57):
As stated earlier, the effects of a task are not made visible to other tasks until the task commits. The overall effect is that tasks appear to operate atomically, but the order in which these atomic operations happen is the order in which the tasks commit, and the commit order is not necessarily the same as the order in which the tasks started.

Detailed Description Text (58):
The OS does not specify the order in which it will start tasks, even in conventional UPA architectures. The OS scheduler takes tasks from a pool of tasks which are ready to run, in an order controlled by relative priorities, system load, fair shares scheduling, and other parameters, and the total ordering is not predictable at the application level.

Detailed Description Text (59):
Order is preserved when there is a causal relationship between tasks. In such cases, the OS waits for a task to commit before starting a task which must follow it. For example, if a process P1 has an ongoing existence, but is run as a sequence of tasks T1, T2, T3, etc., then these tasks must run one at a time in order and the OS must not start one of them until its predecessor has finished. Similar rules apply to messages; if process P1 sends an ordered series of messages to process P2 then it will receive and process those messages in the same order.

Detailed Description Text (62):
Each processor uses its cache to hold copies of locations which are in use by that processor. The data ownership provided by the invention prevents a copy of a location from being held in more than one cache. This avoids the problem encountered in conventional parallel processing architectures of keeping the copies in each cache in step. Data ownership bypasses this problem, because only one PE can own a location. The operation of the cache memory together with the shared main memory is otherwise conventional and will not be described in detail.

Detailed Description Text (63):
Data ownership ensures that a task cannot access any location currently owned by another task (except under some special circumstances described later). This ensures that if two tasks interact by sharing any data, then the effect is exactly as though one of the tasks had committed before the other started. An attempt by a task T2 to access a location owned by a first task T1 causes a collision, which must he resolved before the tasks can proceed.

Detailed Description Text (67):
In the above described Exclusive Access implementation, a task exerts ownership over a memory location upon either a read or a write access to that location. In the second implementation of the invention, a different data ownership mechanism is employed, called Shared Read ownership. This would preferably be used in combination with the Exclusive Access data ownership mechanisms described previously for a certain subset of the shared memory. Selected locations can be defined to be Shared Read locations and to be managed by this mechanism. Shared Read Ownership always enforces logically correct behaviour whatever pattern of read and write access is encountered, but is optimized for the case where a location is read by many tasks and modified only rarely. So long as a Shared Read location is not modified, many tasks can read the location and proceed in parallel. For certain classes of data, such as pointers to data structures this greatly reduces the number of collisions which would otherwise force tasks to rollback. On the other hand, write access to the set of Shared Read locations is strictly limited, and is granted to only one task at a time. Consequently, Shared Read Ownership should be restricted to those locations where only a small minority of tasks need write access; otherwise too many tasks will need to write to Shared Read locations and will be unable to run in parallel.

Detailed Description Text (68):
The decision to define a location as a Shared Read location is made as a part of the

software design process, after study of likely access patterns and intended use.

Detailed Description Text (69):
Every memory location is preferably marked as either Standard or Shared Read under software control when a data structure is initialized, and is not normally changed at other times.

Detailed Description Text (70):
Tasks may read a shared read location without seizing ownership and without being blocked or rolled back. When one or more Shared Read locations must be modified, a writer task seizes ownership of the locations concerned, and other tasks seeking access may then be delayed or blocked.

Detailed Description Text (71):
For clarity of description, a task which is allowed to modify Shared Read locations will be called an Update task. The OS can designate any task as an update task, either when the task is launched or later when the task first attempts to write to a Shared Read location. If a task attempts to write to a Shared Read location, but cannot meet the necessary conditions to become an update task, the OS will force the task to rollback.

Detailed Description Text (72):
During normal system operation, many tasks may read an item of shared read data. Before that item can be safely changed, all these reader tasks must commit or be rolled back. It is not simple to keep records of all tasks which read the data, without incurring a significant time or a space overhead. A simpler solution is to assume that any active task may have read the data. When an update is required, existing tasks are allowed to continue until they commit, or are rolled back, while no new tasks are started. When no active tasks remain, the update task is run alone. After the update task has committed, normal operation is resumed. This solution can be represented as shown in FIG. 10a.

Detailed Description Text (74):
In the improved solution, the update task is allowed to run in parallel with other tasks, but its operation is concealed from the rest of the system. Other tasks can continue to run, and more tasks can be started, but any task attempting to read a Shared Read location owned by the update task is given a copy of the original or rollback data. This means that the update task operation has no detectable effect on the other tasks, which are referred to as "early tasks". At some later time, possibly not until all the update task processing is finished, the number of early tasks is reduced to zero by not allowing any more early tasks to start. When no early tasks remain, the update task is committed, and new tasks called "later tasks" can start.

Detailed Description Text (75):
This improved solution is shown in FIG. 10b. The update task runs before all the early tasks have finished, but does not commit until all these tasks have committed or rolled back. The early tasks may have started before or after the update started, but in any case they logically precede the update, because they cannot access any data modified by the update. All the early tasks must be committed before the update commits. Later tasks start after the update commits, and logically follow the update. The cut off point, after which no more early tasks are started, can occur at any time, but there is little advantage in cutting off early task starts before the update task is ready to commit.

Detailed Description Text (76):
The solution presented above can be improved still further. Once the update task processing is ready to commit it is said to be "complete". The task will not modify any more locations; it already owns every location which it has modified, and it merely waits until all the early tasks have committed or rolled back. This makes it possible to start later tasks as soon as the update is complete, without waiting for it to commit. Of course, later tasks must not read any data modified by the update task before the update task commits, but all such data is already owned by the update task at this point, so it is simple to prevent illegal access. This gives almost complete overlap of early and later tasks, as shown in FIG. 10c.

Detailed Description Text (77):
In summary, the update task divides the universe of tasks into two subsets: the `early` tasks which logically precede the update, and the `later` tasks which start after the update is complete.

Detailed Description Text (82):
2. Block the read, and the reader task, until the update commits, then allow the read to succeed and the reader to continue.

Detailed Description Text (83):
3. Block the read, and force the reader task to rollback.

Detailed Description Text (84):
Choices 1 and 2 allow a greater degree of parallel operation, but both options require the OS to maintain lists of blocked tasks, and become complicated if for some reason the update cannot be committed.

Detailed Description Text (85):
Choice 3 forces a rollback which could perhaps have been avoided, but the mechanism is simple and robust. It affects only a minority of tasks, those which attempt to read a Shared Read location shortly after it has been updated.

Detailed Description Text (86):
A task can enter update mode by making a special call to the OS. Alternatively a trap to the OS occurs if a task not in update mode attempts to write to Shared Read data. In either case, if permission is granted the OS sets certain control bits to allow the task to write to Shared Read locations; otherwise the OS will force the task to rollback.

Detailed Description Text (87):
The update task is then said to be in progress until it indicates that it is complete by making another call to the OS. When the update is complete the OS will flush any changed data to memory but will not yet commit the changes; the update task is now in completed mode. The task is still an update task and will be until the task is committed, as discussed below. The OS does not allow any more early tasks to be started and begins to schedule `later` tasks as described previously. More details of these mechanisms are described later.

Detailed Description Text (90):
Once an update task is complete it must remain in complete but uncommitted mode until all the early tasks have committed or rolled back. Meanwhile SOS can start later tasks and can even allow later tasks to commit provided that they do not read or write any of the data still owned by the complete but uncommitted update.

Detailed Description Text (92):
The protocol described ensures that the update tasks cannot overlap. It also ensures that the update tasks commit in turn, so that the first to operate is the first to commit. This follows, because an update task is not able to commit until all tasks started before it have committed.

Detailed Description Text (93):
Every task is allocated a number called its Generation number, and is said to be a member of that Generation. A global variable called the Current Generation number is maintained by the OS and defines the Current Generation. Every newly started task is a member of the Current Generation. The OS passes this generation number to the SST memory where it is stored in the relevant record. If there are no update tasks, the Current Generation number remains unchanged, and all tasks are members of the Current Generation.

Detailed Description Text (94):
Eventually, some task will become an update task, and it can then write to some Shared Read locations. Only one update task can be in progress at any one time and it will be a member of the Current Generation. In due course the update task will notify the OS that it is complete. At this time, SOS increments the Current

Generation number by one. All tasks started subsequently will have the new Generation number. This also applies to any further update tasks which may be started.

Detailed Description Text (95):
The requirements of the update protocols map naturally into the Generation numbers. There is at most one update task in any Generation, because a new update cannot start until the previous update is complete and the Current Generation number is incremented.

Detailed Description Text (96):
If an update task is a member of generation G, then any other task with generation less than or equal to G is an `early` task; in relation to the update; that is, it started before the update task in generation G was complete. Otherwise it is a `later` task, which started after the update task in generation G was complete. When a task accesses a location owned by the update task in generation G, the accessing task generation is compared with G. If the accessing task has a generation less than or equal to G it is an early task and is allowed to read the original or rollback data from the location. If the accessing task has a generation greater than G, it is a late task, access is denied, and the accessing task must rollback.

Detailed Description Text (98):
Under normal circumstances, relatively few generations will exist at any one time. Typically, at most two or three new update tasks will start before the first commits. Each completed but uncommitted update task remains to in waiting, occupying a task slot. In the example described, there are at most 256 slots, so this sets an absolute maximum to the number of generations which can exist at the same time, but this grossly exceeds the number of generations actually required.

Detailed Description Text (100):
Now a practical implementation of the above shared read protocol will be described with reference to FIGS. 11-13. FIG. 11 is a block diagram of the memory ownership control hardware which is very similar to FIG. 2, but which has the additional features required for Shared Read Access.

Detailed Description Text (101):
The record structure for a record in the tag memory 27 is shown in FIG. 12. Two additional fields are required for the Shared Read implementation, these being the mode field, and the switch field. The Mode field is "0" if the line is in "exclusive access" mode, and is "1" if the line is in "shared read" mode. The Switch field is "0" if the line should remain in the same mode if the owner commits, and is "1" if the line's mode should be switched if the owner commits. With the Shared Read implementation, each individual memory location is individually selectable to be in Shared Read mode or Exclusive Access mode, the particular mode being determined by the Mode field.

Detailed Description Text (103):
Referring now to FIG. 11, only the additional features necessary for the Shared Read Access implementation will be described. The SST memory 26 has an additional output consisting of the Generation number 80 for the SID 81 currently being input. This is fed directly into a generation compare block 82, and also through a latch 83 to the generation compare block 82. The SID 81 may come from either the SID 34 from the OS as before, or may be the SID output 54 from the TST memory 28. A multiplexer 84 selects which SID to use on the basis of a check generation signal 85 generated by the OCL 29. It takes the SID 34 unless the check generation signal 85 is true. The OCL has another additional output consisting of a status signal 87 which is passed to the OS, as described below.

Detailed Description Text (104):
When an access to a Shared Read location is first attempted, the SID 34 from the OS for the accessing task is used as SID input 81. The SST memory 26 outputs the generation number 80 of the accessing task and this is stored in the latch 83. At the same time, the tag memory 28 is checked to see if the location is owned by an active task. If it is, then the SID of the owning task is looked up in the TST memory and output 54 by the TST memory. This SID 54 is then fed back to the SST

together with a "check generation" signal 85 from the OCL 29. The SST memory looks up the generation number of the owning task and outputs this again. The generation numbers of the owning task and of the accessing task (previously latched in latch 83) are compared in the generation compare block 82, and an indication of whether the accessing task is an Early task or a Late task is passed to the OCL. When the mode indicates that the location being accessed is a Shared Read Access location, the Early/Late indication is used together with the Active Copy bit to determine which copy, if any, to return to the accessing task. If the accessing task is Early, then the non Active copy is returned. If the accessing task is Late, then the Active copy is returned and the status signal 87 is output to the OS. When the OS receives the status signal, it knows that it must rollback the accessing task if the relevant update task has not yet completed.

Detailed Description Text (105):
For Shared Read locations, the take ownership command 48 is only issued when a task is granted write access to a location. For Exclusive Access locations, the ownership command 48 is issued for both read and write accesses to a location.

CLAIMS:

1. A parallel processing/shared memory system comprising:

a plurality of processors for running a plurality of tasks each identifiable by a task identifier;

one or more memory modules each having a plurality of memory locations, each memory location associatable with one of the task identifiers;

means for allowing or denying a particular task to access a particular memory location on the basis of the task identifier associated with that location and the task identifier of the particular task, and for associating the task identifier of the particular task with the particular memory location when the particular task is allowed access to that location, and for de-associating the task identifier of the particular task with the particular memory location after the particular task is completed or otherwise terminated thereby making the particular memory location and its contents accessible by another task.

6. A parallel processing/shared memory system comprising:

a plurality of processors for running a plurality of tasks each identifiable by a task identifier;

one or more memory modules each having a plurality of memory locations, each memory location associatable with one of the task identifiers;

means for allowing or denying a particular task to access a particular memory location on the basis of the task identifier associated with that location and the task identifier of the particular task, and for associating the task identifier of the particular task with the particular memory location when the particular task is allowed access to that location;

wherein the memory module has a tag field for each memory location and a data field for each memory location, the tag field including an ownership field for storing ownership information identifying the associated task, the associated task being the owner task for that memory location; wherein the data field comprises a first copy field and a second copy field, and the tag field further comprises an active copy field identifying one of the copy fields as containing an active copy and identifying the other of the copy fields as containing a rollback copy.

11. A system according to claim 9 wherein the tag field further comprises a dirty field, the system further comprising:

a state table for identifying the state for each TIN as either ACTIVE or COMMIT or ROLLBACK, the ACTIVE state defining a task which is currently running, the COMMIT state defining a task which has finished, and the ROLLBACK state defining a task

which is to be rolled back;

wherein:

1) when a new TIN is started, the state in the state table is updated to be ACTIVE;

2) when a TIN is finished, the state in the state table is updated to be COMMIT;

3) when a TIN is to be rolled back, the state in the state table is updated to be ROLLBACK;

4) any task is allowed access to a location which is unowned;

5) when a task having a first TIN attempts to access a location having a second TIN, the state table is consulted for the state of the second TIN; the task is allowed access if the state of the second TIN is COMMIT or ROLLBACK;

6) when a task is first allowed read or write access, its TIN is written into the tag field;

7) when a task is first allowed write access, the dirty field is set.

17. A parallel processing/shared memory system comprising:

a plurality of processors for running a plurality of tasks each identifiable by a task identifier:

one or more memory modules each having a plurality of memory locations, each memory location associatable with one of the task identifiers;

means for allowing or denying a particular task to access a particular memory location on the basis of the task identifier associated with that location and the task identifier of the particular task, and for associating the task identifier of the particular task with the particular memory location when the particular task is allowed access to that location;

wherein the memory locations include standard locations and shared read locations.

18. A system according to claim 17 wherein any task is allowed read access to a shared read location, but only a single task is allowed write access to a shared read location.

19. A system according to claim 18 wherein a task which is allowed write access to a shared read location owns that location and becomes an update task;

tasks which do not access the shared read locations are allowed to continue normally;

tasks which access the shared read locations before the update task completes must commit before the update task;

tasks which access the shared read locations after the update task completes must commit after the update task.

20. A system according to claim 19 wherein a global variable tracks a generation number, and each task is assigned the generation number, and the generation number is incremented each time an update task starts, and thereby allowing only a single update task of a given generation task to exist.

WEST

Generate Collection | Print

L64: Entry 16 of 28 File: USPT Mar 1, 1994

DOCUMENT-IDENTIFIER: US 5291614 A
TITLE: Real-time, concurrent, multifunction digital signal processor subsystem for personal computers

Abstract Text (1):
A personal computer system includes a digital signal processor (DSP) subsystem that is connectable to a plurality of application specific hardware devices. A single DSP is operable under a DSP real-time operating system (RTOS) to concurrently handle a plurality of different signal processing functions on a real-time basis. A DSP data store is connected to the DSP and to the personal computer and includes addressable locations that emulate addressable I/O registers associated with the application specific hardware devices to enable the personal computer to run a plurality of application programs controlling operation of the hardware devices. Performance is enhanced for I/O read and write operations by delaying halting of the DSP allowing such operations to complete in a cycle during which the DSP is not accessing the data store.

Brief Summary Text (2):
This invention relates to the field of data processing, and, more particularly, to an improved, general-purpose digital signal processor (DSP) subsystem for a personal computer, which subsystem is capable of concurrently executing multiple, unrelated signal processing functions on a real-time basis to emulate multiple hardware adapters.

Brief Summary Text (4):
U.S. Pat. No. 4,991,169- Davis et al, for "REAL-TIME DIGITAL SIGNAL PROCESSING RELATIVE TO MULTIPLE DIGITAL COMMUNICATIONS CHANNELS", assigned to the assignee of the present invention, discloses a DSP subsystem in which two single threaded DSPs operate in parallel executing similar or related functions. Each DSP uses an instruction set similar to that of the present invention. Each DSP further has a cycle steal mode of operation (see column 11 of patent) in which DSP operations are halted allowing data to be transferred between the DSP and a personal computer (PC). The present invention is designed to improve upon such prior art system by handling multiple dissimilar functions and improving data transfer efficiency by delaying DSP halting for a preset number of cycles in expectation that certain DSP operations will occur to allow the data transfer to complete without halting.

Brief Summary Text (6):
Digital signal processing is a specialized form of data processing in which digitally represented signals are subjected to rapid, mathematically intensive, repetitive operations where speed is of prime importance. Such operations are executed in a DSP that is specifically designed to execute mathematical algorithms. A DSP commonly includes an arithmetic logic unit (ALU) and a parallel multiplication unit for performing mathematical operations in a single cycle. A DSP may also be supported by both a data store and an instruction store that are accessed in parallel over separate busses to simultaneously transfer both data and instructions so as to avoid memory and bus bottlenecks.

Brief Summary Text (7):
Digital signal processors are used for functions requiring very fast manipulation of numbers, as opposed to data transfers, string operations, or data block handling. A DSP is commonly dedicated to a specific purpose made up of plural tasks. Examples of tasks include: a) Echo cancellation in modem applications; b) Finite impulse

response and infinite impulse response filters with fixed coefficients; c) Adaptive filters with time-varying coefficients; and d) Fast Fourier transforms.

Brief Summary Text (13):
One of the features of the invention is that it is capable of handling multiple digital signal processing functions at the same time up to the bandwidth of the signal processor, and such functions do not necessarily have to be similar. While this design can be dedicated to a single function (such as a CCITT V.32 standard modem engine), it can also be organized to provide multiple functions, e.g., slower speed V.22bis modem operations simultaneously with compressed voice playback. Some functions facilitated by this invention include modem communications at various speeds, speech and audio input and output, data and audio compression and decompression according to numerous standards, and encryption of data using various standards.

Brief Summary Text (14):
Another important feature of the invention is the provision of a real-time operating system (RTOS) which allows for a zero-frame count, enabling rapid changes in control which would not otherwise be available in a priority-based scheduling system. RTOS is composed of a scheduler, an interrupt handler, a queue manager, a task manager, a loader (for loading and patching code), and other assorted subsections. RTOS includes improvements in ordering of tasks through the creation of a 0-order to move certain important tasks to the highest priority, and intertask communications to allow multiple emulated ports to talk with each other, for example.

Brief Summary Text (20):
Another object of the invention is to provide a DSPSS having a DSP and other hardware which allows plural application specific interfaces to be connected to the DSPSS so that different digital signal processing functions can be executed concurrently on a real-time basis.

Brief Summary Text (22):
Another object of the invention is to perform separate functions compatibly without duplicating all previous hardware. "Compatibly" is defined as operating in a fashion transparent to pre-existing application code and attaching to pre-established external electrical and mechanical interfaces.

Brief Summary Text (23):
A further object of the invention is to operate autonomously of a main system processor through a separate memory to provide buffering of I/O communications for more flexible data handling within the system.

Detailed Description Text (2):
Referring now to the drawings, and first to FIG. 1, there is shown an exemplary data processing system comprising a personal computer 10 operable under an operating system to execute application programs. Computer 10 comprises a microprocessor 12 connected to a local bus 14 which, in turn, is connected to a bus interface controller (BIC) 16, an optional math coprocessor 18, and a small computer system interface (SCSI) adapter 20. Microprocessor 12 may be one of the family of 80xxx microprocessors, such as an 80386 microprocessor, and local bus 14 includes conventional data, address, and control lines conforming to the architecture of such processor. Adapter 20 is also connected to a SCSI bus 22 to which is connected a SCSI hard drive (HD) 24 designated as the C:drive, the bus also being connectable to other SCSI devices (not shown). Adapter 20 is also connected to a non-volatile random access memory (NVRAM) 26 and to a read only memory (ROM) 28.

Detailed Description Text (3):
BIC 16 performs two primary functions, one being that of a memory controller for accessing a main memory 30 and a ROM 32. Main memory is a dynamic random access memory (RAM) that comprises a plurality of single, in-line, memory modules (SIMMS) and stores application programs 31 for execution by microprocessor 12 and math coprocessor 18. ROM 32 stores a power on self test (POST) program 33. POST program 33 performs the primary test, i.e. POST, of the system when computer 10 is restarted by turning the power on or by a keyboard reset. An address and control bus 36 connects BIC 16 with memory 30 and ROM 32. A data bus 38 connects memory 30 and ROM

32 with a data buffer 34 that is further connected to data bus 14D of bus 14. Control lines 40 interconnect BIC 16 and data buffer 34.

Detailed Description Text (4):
The other primary function of BIC 16 is to interface between bus 14 and an I/O bus 42 built in conformance with Micro Channel (MC) architecture. Bus 42 is further connected to a video subsystem 44 and to an input/output controller (IOC) 48. Subsystem 44 includes a display 46. IOC 48 controls operation of plurality of I/O devices including a floppy disc drive 50 designated as the A:drive, a printer 52, and a keyboard 54. IOC 48 also is connected to a mouse connector 56, a serial port connector 58, and a speaker connector 60, which allow various optional devices to be connected into the system.

Detailed Description Text (5):
Bus 42 is also connected to a plurality of MC connectors 62. A DSPSS 64 can be configured as a feature card 66 that is plugged into one of connectors 62 and is thereby connected into the system through bus 42. Obviously, the DSPSS could also be connected directly to bus 42 or to bus 14. Card 66 has a plurality of ports which are respectively connectable by a plurality of cables 74, 76 and 78 to three application specific hardware (ASH) applications 80, 82 and 84. Such hardware would obviously be installed by the user of the system to perform desired digital signal processing functions or applications. DSPSS

Detailed Description Text (6):
Referring to FIG. 2, DSP card 66 includes a DSP chip 90 mounted on the card, which in turn includes a DSP 92 and associated circuitry. The components that are mounted on the card directly are those located outside of the rectangle in FIG. 2 which represents chip 90 and inside of the lines representing card 66. Timing for the DSP is provided by an oscillator 93 and a phase generator 94 which divides the basic cycle of the oscillator into four phases. The data lines 42D of I/O bus 42 are connected to a chip data bus 95 through two way drivers 96 and latches 100. Incoming data is driven onto bus 95 over lines 98 and outgoing data is latched in latches 100 and driven from lines 102 onto data lines 42D.

Detailed Description Text (7):
A RAM data store 104 is connected to bus 95 for storing information including various data structures described hereinafter and the operands or data being read into and written from DSP 92. An instruction store 106, including a RAM 108 and a ROM 110, is connected by a bus 112 to DSP 92. The data store and the instruction store are thus arranged in the manner of the well known Harvard architecture for a DSP system, wherein data and instructions are concurrently transferred and fetched under the control of DSP 92 in parallel on separate, independent busses. Bus 112 is operated and controlled primarily by the DSP while bus 95 is shared by DSP 92 and processor 12 through bus 42. Programs for RAM 108 are first stored in data store 104 from bus 42D and then transferred or loaded from store 104 into RAM 108 for execution by DSP 92.

Detailed Description Text (8):
A memory control circuit 113 is connected to busses 95 and 112 to control accessing data store 104 and instruction store 106. Circuit 113 is also connected by lines 114 to phase generator 94 to allow coordination of the phase generation with accessing the stores and control of the phase generator for an extended cycle. Generator 94 can stretch clock cycles to accommodate slower ROM instruction fetching. Circuit 113 is also connected by lines 115 to a register array 116 to control accessing registers 117-123 in the array. The array is also connected to data bus 95 for transferring data into and out of the registers via bus 95. Registers 117-123 include appropriate conditioning circuits and store different data or information for different functional purposes as described hereinafter.

Detailed Description Text (10):
DSP 92 is interrupt driven and supports both simple processing with a minimum amount of interrupt overhead and fully asynchronous task processing with complete saving and restoring of the processor resources by software. DSP 92 is a parallel and pipelined horizontal processor which achieves maximum throughput of up to three distinct operations per cycle time. The operations include a memory transfer, an

arithmetic or logical computation, and a multiplication.

Detailed Description Text (11):
Data bus 95 includes address lines that are shared by DSP 92 and I/O bus 42. When one of these elements drives the address lines of bus 95, the other is prevented from doing so. When bus 42 drives bus 95, control thereof is managed by sequencer 128. Bus 95 also includes data lines that are shared by I/O bus 42 and DSP 92. Data is transferred to and from DSP 92 for program loading of instruction store RAM 108 and to and from bus 42. Bus 112 also includes address and data lines, the latter being used primarily by the DSP to fetch instructions from instruction store 106. The address lines of bus 112 are used by DSP 92 for the selection of instructions to be transferred from store 106 into DSP 92.

Detailed Description Text (12):
Chip 90 also includes external digital control and analog interface support logic and logic for interfacing with bus 42, the former logic including modem control register 118, an analog interface control (AIC) shift register 117, AIC and analog logic 134, and TTL drivers 132. Logic 134 and drivers 132 respectively handle analog and serial digital signals, and are connected to a bus 135 that in turn is connected to cables 74, 76, and 78 through connectors C which form three ports for connection to the ASHs. Bus 135 carries both analog and digital signals where the specific signals used for any one port are determined by the specific pinout or line connections. Thus each port can be used as an analog port or as a serial digital port as desired. Modem control register 118 holds control and status bits, and register 117 is a shift-left and a parallel load-and-read register under the control of signals from AIC 134.

Detailed Description Text (13):
Arbitration register 119 is connected through drivers 131 to the arbitration lines 42AR of bus 42 and contains the arbitration information required by bus 42 for use in DMA transfers. When a DMA transfer is required, register 119 is loaded with an arbitration level and a DMA request bit. At the completion of arbitration, the request bit is reset. A bus control register (BCR) 120 is connected through drivers 130 to the bus interrupt lines 42I of bus 42. Register 120 contains latches for setting and resetting the interrupt request line and the channel check line of bus 42 under control of the DSP program. It also includes status bits indicating word or byte transfer and reflecting the condition of DMA controller terminal count, and a RETURN NOT READY bit that is automatically set by any decoded write command or a read command when an interrupt on read is enabled. It also includes a clock control bit for stretching the clock to accommodate slower ROM instructions.

Detailed Description Text (16):
Comparator 124 compares the emulation port address from BDAR 122 with the address on lines 42A of bus 42. If the address on lines 42A is in the range specified in BDAR 122, data is transferred as a read or write operation. Translator 126 converts I/O addresses from bus 42A into addresses in data store 104 and is controlled by sequencer 128 for gating the converted address onto the output address bus. Translator 126 is initialized by storing therein tables for mapping the I/O addresses into data store addresses.

Detailed Description Text (18):
DSP 92 is similar to the DSPs described in the above mentioned U.S. Pat. No. 4,991,169 and is designed to execute an instruction set of the same kind, so that only a brief description is provided herein. Referring to FIG. 3, bus 95 forms in DSP 92 a common data bus (CDB) having a sixteen bit wide data path for transferring data around the DSP. DSP 92 includes an arithmetic logic unit (ALU) 136 for performing adding, logic, and shifting operations, and a multiplication unit (MPY) 138 for performing multiplication in a single cycle. ALU 136 receives "A" and "B" operands, and MPY 138 multiplies "X" and "Y" operands to produce a thirty two bit result.

Detailed Description Text (19):
The remaining items shown in FIG. 3 are principally concerned with moving the various operands into, around and out of ALU 136, MPY 138 and DSP 92. More specifically, an LMUX 140 and a RMUX 142 have inputs connected to CDB 95 and to a

saturation (SAT) control circuit 146 which receives the sum or results from ALU 136. Thus operands from either 95 or 136 can be moved through MUXes 140 and 142 into a stack 144. The general functions of stack 144 are to input "A" operands into ALU 136, place results on CMUX bus 102, feed an operand to the address generator discussed with reference to FIG. 4, and feed an "X" operand to MPY 138. Stack 144 comprises a plurality of paired, general purpose registers R0-R7 from which operands can be selectively fed into an AMUX 150 and a CMUX 152. The output of 150 is fed through a gate 154 into ALU 136. The output of CMUX 152 if fed through a driver (D) 156 onto bus 102. Registers R0 and R4 are connected by lines 158 to address generator 212 (FIG. 4). Register R5 is connected by lines 160 to the "X" operand input of MPY 138.

Detailed Description Text (20):
A plurality of status registers 162-166 have their outputs connected through a plurality of drivers 168 to bus 102. Register 162 receives a RIPA operand from instruction decode register (IDR) 112D. Registers 163-166 respectively store RCDB, MCRH, MCRL, and PSRH operands which are inputted from bus 95. Register 166 is connected to a MUX 170 which has one set of inputs connected to bus 95 and other inputs connected to lines 172. MPY 138 has its "Y" operand input connected to bus 102 so that the "Y" operand can come from a plurality of different sources. The output of MPY 138 is fed into MUXes which in turn have second inputs connected to bus 95. MUXes 174 feed a thirty two bit product register 176 having four outputs connected to inputs of MUX 148. MUX 148 also has inputs connected to lines 178 and to bus 102. The output of MUX 148 is fed via lines 180 to the "B" operand input of ALU 136.

Detailed Description Text (21):
A pair of drivers 182 and 184 are respectively connected at their inputs to bus 102 and lines 180 and drive signals therefrom onto bus 95. Line 180 is also connected to an input into MUX 186 along with other inputs from lines 178, rounding bit RD (multiply result of zero, overflow, negative, positive), and an increment of (+1). A gating signal CIE controls which input is gated from MUX 186 into ALU 136 to handle carries. The ALU is also connected to lines 178 to feed signals to MUX 174 and to a MUX 188. MUX also has an input connected to lines 178 and produces an output that is fed to the branch and decode logic.

Detailed Description Text (22):
Referring to FIG. 4, the instruction sequencing and control portion of DSP 92 comprises an instruction address register (IAR) 190 connected to the address bus 112A of bus 112 so as to transmit instruction addresses to instructions store 106. Instructions are transmitted on data bus 112D of bus 112 and stored in an instruction data register (IDR) 192. IDR 192 is further connected to instruction decode logic 194 which decodes each instruction and sets up an execute register 196 so as to generate control signals appropriate to executing each instruction as it is decoded. IAR 190 receives each instruction address through an instruction address (IADDR) MUX 198 under the control of a selection logic (SEL LOG) circuit 200 that handles interrupts and branching. Circuit 200 receives control signals on a line 202 from interrupt control logic (ICL) 204 and on a line 206 from branch decode logic 208. MUX 198 receives addresses from bus 95, ICL 204 via line 210, an address generator (ADDR GEN) adder 212 via line 214, and an adder 216 by line 218. Adder 212 increments the address outputted by IAR 190 and feeds the new address back into IAR 190 through MUX 198.

Detailed Description Text (23):
Adder 212 has two operand inputs "A" and "B" which receive operands respectively from an operand selector 220 and an index MUX 222. Selector 220 is connected to receive operands from IDR 192. MUX 222 receives operands selectively from an instruction link register (ILR) 224, registers R0 and R4 and register extensions R0EXT and R4EXT. ILR 224 is connected to the output of IAR 190. Adder 212 is further connected by line 226 to index decode logic 228 which allows indexed jumps. Line 214 is further connected to transmit signals thereon to a common address bus register (CABR) 232 and to R0EXT and R4EXT. CABR 232 has its outputs connected to a driver 232 for driving signals onto bus 95, and to data store 104 via the address bus 102A of bus 102. Data store is further connected to by data bus 102D to drivers 234 and 236 providing a two way data transmission between busses 95 and 102.

Detailed Description Text (24):
ICL 204 includes a processor state register low (PSRL) 240 which receives data from PSRL write logic 242 and outputs data via a driver 244 onto CMUX bus 102. Logic 242 is connected to bus 95 to receive data therefrom. ICL further receives as inputs interrupt signals INT0-INT7, a power on signal POWR, and a multiply product overflow signal PROR. The interrupt control is very important to concurrently executing plural functions and is discussed in greater detail hereinafter. ICL 204 is connected by line 246 to a latch 248 having an output further connected into logic 194. A second latch 250 is also connected to logic 194 and has an input from line 252 which is connected to the output from logic 208 to provide feedback therefrom. IDR 192 is connected by lines 254 as inputs into each of logics 194,228, and 208. Logic 228 also receives an input signal from MCR 164, 165.

Detailed Description Text (26):
Referring to FIG. 5, when the PC issues an I/O read or write command to the DSPSS, step 260 makes a comparison between the I/O address from bus 42 and the address ranges in the DSPSS to detect valid addresses therein. In response to detecting a valid address, step 262 then gates a "select feedback" signal and a "not ready" signal to the bus in accordance with the Micro Channel architecture. In step 264, so long as a "return not ready" signal is active, step 264 loops on itself until such signal becomes inactive whereupon step 266 sets a "wait count" to zero. Step 268 then determines if the "wait count" has reached a preset value of seven to establish a maximum number eight of delayed cycles before the DSP will be halted. So long as the "wait count" is not seven, step 268 branches to step 270 which checks to see if the "DMA acknowledge" signal is active. If it is not, then a branch is made to step 272 which increments the wait count by one and branches back to step 268 to establish a loop that so long as the DMA acknowledge signal is inactive continues until the wait count equals seven and then a branch is made to step 274. Step 270 provides cycle timing for incrementing the wait count once each DSP cycle. During the course of a DSP cycle, which as previously indicated is divided into four phases, a determination is made e.g. in phase three as to whether the next cycle will involve the data store. If the data store is not to be used on the next cycle, the DMA ACK signal is set active and control passes to 278 to complete the data transfer in the next cycle without halting the DSP.

Detailed Description Text (27):
Step 274 sends a "DMA request" signal to the DSP requesting a halt. Until the DSP is halted, the DMA acknowledge signal remains inactive and step 276 loops back to step 274. When the DSP halts, and a cycle steal occurs, the DMA acknowledge signal is activated, and control passes through step 270 to step 278 which gates the address bus in. For an I/O write command, step 280 then gates the data in for writing into the location designated by the I/O address. Step 282 then gates the IWCR bits 0,1 to the DSP for read/write control to the data store. Steps 284 and 286 then complete the bus interaction by gating the "gate ready" signal to the bus and dropping the "select feedback" signal. During a read operation, step 288 gates the IWCR bits 0,1 to the DSP and step 290 gates the data being read onto the data bus out. Step 290 is then followed by steps 284 and 286. In summary, the delayed halting has several advantages. It provides a greater throughput by not stopping or halting the DSP just to move data to and from the DSP. It also achieves a balanced throughput allowing a greater control of the resources. Program task duration is controlled tighter due to the time the DSP is halted. Real-time processing can be handled because the processor is not stopped, and guaranteed bandwidth is now possible with dissimilar devices.

Detailed Description Text (30):
Data store 104 stores mapped I/O control registers 318, shadow registers, port I/O, set up registers, etc. 320, RTOS data area 322, intertask communication buffers 324, task defined buffers, tables, etc. 326, and task control blocks (TCBs) 328. The shadow registers are addressable locations for emulating the normal addressable registers located in a functional device, the emulation being accomplished by software so as to provide a low cost alternative to using actual hardware registers. Because such registers are emulated by software, they can be reconfigured and altered to suit the requirements of a particular function or changes in the hardware being emulated.

Detailed Description Text (34):
When invoked, TMS runs through the unordered linked list, comparing each task's COUNT against the current COUNT of TMS itself. When the COUNT of the task is found to be equal to that of TMS, it is updated by adding the task's FRAME to it. Then, TMS inserts the task into the prioritized execution queue according to this updated COUNT. TMS sets INTFLAG of the task's TCB equal to -1 to indicate that the task is linked to the execution queue and is ready to go.

Detailed Description Text (36):
When TMS encounters its own TCB in the unordered circular list, it branches to end TMS operation for this processing frame. The least updated COUNT value of all tasks on the TMS unordered list is taken as the next COUNT of TMS itself.

Detailed Description Text (37):
A typical RTOS environment may include several functions running concurrently, each comprised of a moderate to a large number of tasks and subtasks. Usually, most of them will be synchronous real-time tasks which can be grouped into equal FRAME and initial COUNT classes of synchronous tasks. All of these tasks are linked to the TMS task. This would create a severe loading of the processor each time such a group has to be placed into the execution queue, since TMS load is proportional to the square of the number of tasks being inserted.

Detailed Description Text (56):
TMSNEXT Pointer to next task's TCB on Task Manager's list

Detailed Description Text (59):
COUNT The current count for the task. This count indicates the absolute count for the next scheduling the task for execution. The Task Manager adds FRAME to the count each time the task is scheduled. For example, to schedule a task for execution on the first sample clock the initial count value would be 5. Subsequent scheduling is dependent on the FRAME value.

Detailed Description Text (60):
INTFLAG Used by the Task Manager to indicate the interrupt state of the Task. A task is either idle, interrupted, or on the execution queue.

Detailed Description Text (61):
STATE Used by the Task Manager to indicate whether a task is active, requested to become inactive (transient), or inactive. There are three distinct states that a task can be in when resident in the DSP subsystem. First, a task may be active. An active task has the STATE flag of its TCB equal to zero. Its COUNT is never less than the COUNT of TMS and its COUNT is perfectly divisible by the sample rate scale factor, with no remainder. Second, a task may be active but has been requested to become inactive by another task. In this state, the STATE flag of its TCB is equal to one. This second state is transient. As soon as the task has completed execution within its FRAME time, the background executor sets its STATE equal to two, translating the task into the third distinct state, inactive. The only way for a task to enter the third state is from the second state. There is one major reason why a task must be allowed to complete execution before becoming inactive. It is possible that the task was in an interrupted state when the request came for inactivity. If the task were reactivated at some later time, beginning at the top of its code or restoring from an interrupted state, no guarantee could be made that the task would continue normally. For example, a variable might take on a transient state during a task's block time and then be restored to a steady state. Thus, restarting from the top of the code could cause a problem. Tasks that are inactive can be reactivated by setting the STATE flag of its TCB to zero.

Detailed Description Text (78):
As previously indicated, DSP 90 has eight interrupt signal lines INT0-INT7. Each interrupt line is assigned a predetermined function. RTOS 300 handles all INT0 (sample clock) and INT1 (PC read/write) interrupts. INT0 is driven by clock 334 and is used by the RTOS to schedule tasks for execution. INT1 is driven by the PC system bus. An interrupt INT1 occurs each time the PC reads or writes memory 104. INT1 is used to signal the instant when data is available from or data can be transferred to

the PC application program. Interrupts INT2-INT7 are used to transmit and receive data and are handled by the tasks themselves. Entry points to the task's interrupt handlers are contained in the TCB.

Detailed Description Text (79):
The transmit and receive interrupts are enabled only for those ports that have an interface card installed. When a transmit interrupt occurs, the task is given control at the location pointed to by INTHAND1 of the TCB. For receive interrupts the task is given control at the location pointed to by INTHAND2. The task interrupt handlers get control with the processor in foreground mode. These interrupt handlers must execute an absolute minimum of instructions and return to background, by executing a BLEX instruction, in order not to disrupt the task switching functions of RTOS. At no time are tasks allowed to manipulate the interrupt mask bits of the MCRL. Interrupt level assignments are:

Detailed Description Text (85):
Interrupts on Levels 2 through 7 are handled by the task running on that particular port. The task provides an entry point to its interrupt handlers in the TCB. When a task consist of subtasks the interrupt entry points must be identical in all subtask TCBs.

Detailed Description Text (86):
When an interrupt occurs RTOS saves the ILR at absolute data store location ILRSAVE and then passes control to the task at the entry point specified in the TCB. The port ID is passed in R0. The interrupt handler then processes the interrupt, exits the level, and returns control at the location saved in ILRSAVE.

CLAIMS:

1. A personal computer comprising:

a main memory for storing application programs:

a first processor for executing said application programs;

a plurality of application specific hardware (ASH) devices;

a general purpose digital signal processing subsystem (DSPSS) connected to said ASH devices; and

bus means interconnecting said first processor, said main memory, and said DSPSS;

said DSPSS comprising

a plurality of ports connected to said ASH devices for transmitting analog signals and digital signals between said ASH devices and said DSPSS,

converter means for converting said analog signals into digital operands,

an instruction store connected to said DSP for storing a plurality of signal processing tasks for execution by said DSP,

a data store connected to said DSP and to said bus means so that said DSP and said first processor can independently access said data store, said data store being further connected to said ports and to said converter means for storing said digital operands and said digital signals,

a digital signal processor (DSP) for processing said digital signals and said digital operands, said DSP comprising an arithmetic logic unit and a multiplication unit for executing tasks to process said digital operands and said digital signals, and

control means connected to said DSP for operating said DSP and concurrently executing a plurality of said signal processing tasks in response to execution of said application programs by said first processor;

said control means comprising

a scheduling clock for generating a plurality of scheduling interrupts at fixed time intervals;

a plurality of task control blocks (TCBs) stored in said data store, there being a different TCB associated with each task, each TCB containing a plurality of fields for storing, for the associated task, 1) a frame defining a scheduling period between successive scheduling of said associated task, 2) a scheduling count defining when the associated task is to be next scheduled, 3) DSP information for restoring said DSP when an interrupted task is next executed, and 4) an instruction address at which to begin initial execution of said associated task;

a background executor for maintaining a queue of tasks scheduled for execution;

a foreground executor that is periodically executed in response to said scheduling interrupts for scheduling said tasks, said foreground director including means for 1) saving DSP information from a task being interrupted, 2) searching said unordered list of TCBs to find a TCB whose count field has reached a predetermined value, 3) adding the task associated with such TCB, to said queue of tasks, 4) updating said count field, in said TCB of the task added to said queue, by adding said frame to said count, 5) ordering said tasks in said queue according to which tasks have the lowest counts in said TCBs associated therewith, with the task having the lowest count being at a head of said queue, and 6) passing control to said background executor;

said background executor including means, operative in response to receiving control from said foreground executor, for passing control to said task at said head of said queue for execution thereof, for removing said task from said queue upon completion of execution thereof, and for passing control to the next task in line in said queue when another task has completed execution.

4. A personal computer in accordance with claim 1 wherein said DSPSS further comprises:

a cycle counter;

analyzing means for determining whether or not said DSP is going to access said data store in an immediately succeeding cycle;

incrementing means for incrementing said counter when said analyzing means makes a positive determination;

means, operative in response to said counter reaching a predetermined count, for halting operation of said DSP after delaying for a number of cycles defined by said predetermined count, to thereby allow said data store to be accessed by;

and means for granting access to said data store in response to I/O commands from said first processor, when said analyzing means make a negative determination and when said DSP is halted.

WEST

Generate Collection Print

L64: Entry 27 of 28 File: USPT Mar 6, 1984

DOCUMENT-IDENTIFIER: US 4435752 A
TITLE: Allocation of rotating memory device storage locations

Drawing Description Text (4):
FIG. 3 diagrammatically illustrates timesharing of virtual processors in the peripheral processor of FIGS. 1 and 2;

Drawing Description Text (5):
FIG. 4 is a block diagram of the peripheral processor;

Drawing Description Text (7):
FIG. 6 illustrates the virtual processor sequence control of FIG. 4;

Drawing Description Text (11):
FIG. 11 illustrates the interfaces involved for driver management from a peripheral I/O request which was initiated to the task controller;

Drawing Description Text (14):
FIG. 14 illustrates the post request processor controls;

Drawing Description Text (16):
FIG. 16 is a block diagram which shows the relationship between the hardware interface and stored program components of the computer system for which the invention applies.

Drawing Description Text (19):
FIGS. DA2-DA4 show the Disc Assignment Pre-Processor.

Drawing Description Text (21):
FIGS. DA10-DA19 show the Disc Assignment Processor routine.

Drawing Description Text (23):
FIGS. DA26-DA32 show the Disc Release Processor.

Drawing Description Text (25):
FIGS. P1-P3 show the entry task of the Open Command.

Detailed Description Text (3):
The memory provides for 140 nanosecond cycle time, and on the average, 120 nanosecond access time. Each memory access results in the transfer of information in groups of 8 32-bit words, hereafter referred to as an octet. Thus, each memory module 12-19 is partitioned into 2048 octets.

Detailed Description Text (7):
It is to be understood that the processor system thus has a memory or storage hierarchy of five levels. The most rapid access storage is in the CPU 34 which has nine octet buffers, each octet consisting of 256 bits. The next most rapid access is in the active element memory units 12-19. The next most rapid access is in the bulk memory extension 49. The next most available storage is the disc storage units 38 and 39. Finally, the tape units 27-32 complete the storage array.

Detailed Description Text (10):
A signal bus 43 extends between the memory control 20 and data channel unit 36 which

is connected to the head per track discs 38 and 39 by way of a disc interface unit 37. Each disc module has a capacity of 25 million words. The data channel unit 36 and the disc interface unit 37 support the disc units 38 and 39. The data channel unit 36 is a simple wired program computer capable of moving data to and from memory discs 38 and 39 through the disc interface unit 37. Upon command only from the PPU 22, the data channel unit 36 may move memory data from the discs 38 and 39 via the bus 43 through the memory ccontrol unit 20 to the memory modules 12-19 or to the memory extension 49.

Detailed Description Text (11):
Bidirectional channels extend between each disc 38 and 39 and the disc interface unit 37. One data word at a time is transmitted between a disc unit 38 and 39 and the data channel unit 36 through the disc interface 37. Data from memory stacks 12-19 are transmitted to and from data channel 36 in the memory control unit 20 in eight-word blocks.

Detailed Description Text (12):
A single bus 41 connects the memory control unit 20 with the PPU 22. PPU 22 operates all I/O devices except the discs 38 and 39. Data from the memory modules 12-19 are transferred to and from the PPU 22 via a memory control unit 20 in eight word blocks.

Detailed Description Text (14):
The PPU 22 contains eight virtual processors therein, the majority of which may be programmed to operate various ones of the I/O devices as required. The tape units 27 and 28 operate a 1" wide magnetic tape, while tape units 29-32 operate with 1/2" magnetic tapes to enhance the capabilities of the system.

Detailed Description Text (15):
The virtual processors in the PPU 22 take instructins from the central memory and operate upon these instructions. The virtual processors include program counters and a time-shared arithmetic unit in the peripheral processing unit. The virtual processors execute programs under the instruction control. The PPU 22 and the virtual processors are described in more detail in U.S. Pat. No. 3,573,852 for "Variable Time Slot Assignment of Virtual Processors," assigned to Texas Instruments Incorporated and incorporated herein by reference.

Detailed Description Text (16):
The PPU 22 operates upon the program contained in memory and executed by virtual processors in an efficient manner and additionally provides monitoring controls to the programs being run in the CPU 34.

Detailed Description Text (17):
CPU 34 is connected to memory stacks 12-19 through the memory control 20 via a bus 42. The CPU 34 may utilize all eight words in an octet provided from the memory modules 12-19. Additionally, the CPU 34 has the capability of reading or writing any combination of those eight words. Bus 41 handles three words every 60 nanoseconds, two words input to the CPU 34 and one word output to the memory control unit 20.

Detailed Description Text (18):
Buses 44-47 are provided from the memory control unit 20 to be utilized when the capabilities of the computer system are to be enlarged by the addition of other processing units and the like.

Detailed Description Text (19):
Each of the buses 41-48 is independently gated through the memory control unit 20 to each memory module 12-19 thereby allowing memory cycles to be overlapped to increase processing speed. A fixed priority preferably is estabilished in the memory controls to service conflicting requests from the various units connected to the memory control unit 20. The internal memory control unit 20 is given the highest priority with the external buses 43, 41, 42, 43 and 44-47 being serviced in that order. The external bus processor connectors are identical, allowing the processors to be arranged in any other priority order desired.

Detailed Description Text (20):

The dual mode bulk memory unit 49 is connected to the memory control unit 20 by means of buses 50 and 48. The maximum data rates over busses 48 and 50 is 40 megawords per second. Data in the dual mode bulk memory unit 49, transferred bia bus 50, is in the address space of the high speed memory modules 12-19, and randomly accessed, 8 words per fetch cycle. Data may be moved to and from the dual mode bulk memory unit 49 via bus 50 in a random access fashion from any processor located on buses 41-48 which includes the bulk memory unit itself. Blocks of contiguous data are moved to and from the dual mode bulk memory unit 49 over bus 48 to and from any memory module 12-19 by contrtol of a data channel built into the bulk memory unit 49. The data channel unit built into the bulk memory unit 49 is initiated by the PPU 22 by communication via bus 40.

Detailed Description Text (21):
In typical operation, programs awaiting execution on the discs 38 and 39 are moved by control of the data channel unit 36 through bus 43 by way of memory control unit 20 and through bus 50 to the dual mode bulk memory unit 49. When storage becomes available in the high speed memory, consisting of modules 12-19, regions of data can be transferred at 40 megawords per second from the bulk memory unit 49 by way of bus 48 under control of the memory control unit 20 to any one of the memory modules 12-19. This action is controlled exclusively by the PPU 22.

Detailed Description Text (22):
The PPU 22 in the present system is able to anticipate the need and supply demands of the CPU 34 and other components of the system generally by utilization of the particular form of control for time-sharing as between a plurality of virtual processors within the PPU22. More particularly, programs are to be processed by a collection of virtual processors within the PPU22. Where the programs vary widely, it becomes advantageous to deviate from unpartial time-sharing as between virtual processors.

Detailed Description Text (23):
In the system shown in FIG. 3, some virtual processors may be greatly favored in allocation of processing time within the PPU22 over other virtual processors by the multiprocessor control system. Further, provision is made for changing frequently and drastically the allocation of time as between the processors.

Detailed Description Text (24):
FIG. 3 indicates that the virtual processors P.sub.0 -P.sub.7 in the PPU22 are serviced by the arithmetic unit AU400 of PPU22.

Detailed Description Text (25):
The general concept of cooperation on a time sharing sense, as between an arithmetic unit, such as unit 400 and virtual processors such as processors P.sub.0 -P.sub.7 is known. The system for controlling such a configuration is descirbed herein. The processors P.sub.0 -P.sub.7 are virtual processors occupying sixteen time slots. The construction of the present system provides for variable control of the time allocations in dependence upon the nature of the task confronting the overall computer system. P.sub.0 is a dedicated processor in which the master controller executes at all times.

Detailed Description Text (26):
In FIG. 3, eight virtual processors P.sub.0 -P.sub.7 are employed in PPU22. The arithmetic unit 400 of PPU22 is to be made available to the virtual processors one at a time. More particularly, one virtual processor is channeled to the arithmetic unit 400 with each clock pulse. The selections from among the virtual processors is performed by a sequencer diagramatically represented by switch 401. The effect of a clock pulse represented by a change in position of the switch 401 is to actuate the arithmetic unit 400 which is coupled to the virtual processors in accordance with the code selected for time slots 0-15. Only one virtual processor may be used to the exclusion of all the others at one extreme. At the other extreme, the virtual processors might share the time slots equally. The system for providing this flexibility is shown in FIGS. 4-6.

Detailed Description Text (27):
FIG. 4. The organization of the PPU22 is shown in FIG. 4. The central memory 12-19